编程笔记

作者：igrape
组织：TropicalTeamYard
时间：2021 年 7 月 10 日
版本：0.1

各人自扫门前雪，休管他人瓦上霜。——陈元靓

特别声明

编程笔记

igrape

2021 年 7 月 10 日

目录

第 1 章 Java&Kotlin

1.1 Joda-time 时间处理类 (since 1.8)

create: 2021/7/10 update:2021/7/10

1.1.1 简介

在 Kotlin 中处理日期和时间是很常见的需求，在 Java 1.8 之前，我们常常使用 **Date** 和 **Calendar** 类来处理日期和时间。然而这些 Api 的功能并不是很常用，而且使用也不太方便，因此，Joda-time 成为 Java 日期和时间处理的事实上的标准。

为了更好的介绍 Joda-time 是如何设计、实现时间的处理的，我们仍然有必要介绍一下 Java 中日期和时间的传统 Api。下表给出了在日期和时间处理中使用到的各个 **class**。

表 1.1: 日期和时间处理中常用到的 **class**

新/旧	class	描述
旧	java.util.Date	表示一个时间节点，计时精度为 ms
旧	java.sql.Date	表示一个日期
旧	java.util.Calendar	日历系统
旧	java.sql.Timestamp	表示时间轴上一个瞬时的点，计时精度为 ms
新	java.time.Instant	表示时间轴上一个瞬时的点，计时精度为 ns
新	java.time.Duration	表示一个时间段，计时精度为 ns
新	java.time.DateTime	表示时间轴上一个时间点，计时精度为 ns
新	java.time.LocalDate	表示当前时区一个时间点的日期部分，计时精度为 ns
新	java.time.LocalTime	表示当前时区一个时间点的时间部分，计时精度为 ns
新	java.time.LocalDateTime	表示当前时区一个时间点，计时精度为 ns
新	java.time.ZonedDateTime	表示一个带时区的时间点，计时精度为 ns

笔记 **java.time.Instant** 是以 UTC（Coodinated Universal Time，世界协调时）来计时的，即 +0:00。

新的日期与时间处理类相比于旧的日期处理类改进了以下几点：

1. 将计时精度从原来的 ms(毫秒) 提高到了 ns(纳秒).
2. 新的 API 为不可变类型，旧的 API 为可变类型，支持作为 **HashMap** 的 key.
3. 能够方便的表示时区以及在各个时区内互相转换.

一般来说，一个完善的日期与时间处理类应当支持以下几个功能：

1. 获取当前时间.
2. 能够表示各个时区的同一个时间.
3. 能够获取当前时间的各种属性.
4. 能够支持添加或者减少一个时间段.
5. 完善的格式化.

1.1.2 时区与偏移

时间标准

在将时区与偏移前我们需要先将一个概念：时间标准。[1]

定义 1.1. GMT

GMT（Greenwich Mean Time），格林威治平时，它规定太阳每天经过英国伦敦郊区的皇家格林尼治天文台的时间为中午 12 点。
格林威治皇家天文台为了海上霸权的扩张计划，在十七世纪就开始进行天体观测。为了天文观测，选择了穿过英国伦敦格林威治天文台子午仪中心的一条经线作为零度参考线，这条线，简称格林威治子午线。
1884 年 10 月在美国华盛顿召开了一个国际子午线会议，该会议将格林威治子午线设定为本初子午线，并将格林威治平时 (GMT, Greenwich Mean Time) 作为世界时间标准（UT, Universal Time）。由此也确定了全球 24 小时自然时区的划分，所有时区都以和 GMT 之间的偏移量做为参考。
1972 年之前，格林威治时间（GMT）一直是世界时间的标准。1972 年之后，GMT 不再是一个时间标准了。

♣

定义 1.2. UTC

UTC（Coodinated Universal Time），协调世界时，又称世界统一时间、世界标准时间、国际协调时间。由于英文（CUT）和法文（TUC）的缩写不同，作为妥协，简称 UTC。
UTC 是现在全球通用的时间标准，全球各地都同意将各自的时间进行同步协调。UTC 时间是经过平均太阳时（以格林威治时间 GMT 为准）、地轴运动修正后的新时标以及以秒为单位的国际原子时所综合精算而成。
在军事中，协调世界时会使用“Z”来表示。又由于 Z 在无线电联络中使用“Zulu”作代称，协调世界时也会被称为"Zulu time"。

♣

时区

正是由于 12:00 时刻太阳在正上方这个观念，再加上同一时刻地球上不同的区域对应的太阳位置是不一致的。时区与偏移的概念便因此而生。

定义 1.3. 时区

从格林威治本初子午线起，经度每向东或者向西间隔 15°，就划分一个时区，在这个区域内，大家使用同样的标准时间。
但实际上，为了照顾到行政上的方便，常将 1 个国家或 1 个省份划在一起。所以时区并不严格按南北直线来划分，而是按自然条件来划分。另外：由于目前，国际上并没有一个批准各国更改时区的机构。一些国家会由于特定原因改变自己的时区。
全球共分为 24 个标准时区，相邻时区的时间相差一个小时。

♣

例如我们现在有两个时区 **Asia/Shanghai(+8:00)** 和 **Asia/Tokyo(+9:00)**。同样一个时间 **2021/1/1 10:00+0:00**，在 Shanghai 就是 **2021/1/1 18:00+8:00**，而在 Tokyo，就是 **2021/1/1 19:00+9:00**。这种差异正式因为 Shanghai 与 Tokyo 的经度差异约在 15° 导致的。

地球自转一天 24 个小时，每个小时的区域划分一个时区，我们就可以得到 24 个时区，每个时区的跨度约为 15°，当然，为了防止出现一块小岛或者陆地被划分成两个区域，也会人为地调整时区的分割。时区图可以由图1.1表示。[2]

[1]参考自文章：https://www.cnblogs.com/champyin/p/12767852.html

[2]图片来源：http://m.shijieditu.net/world/timezone.html

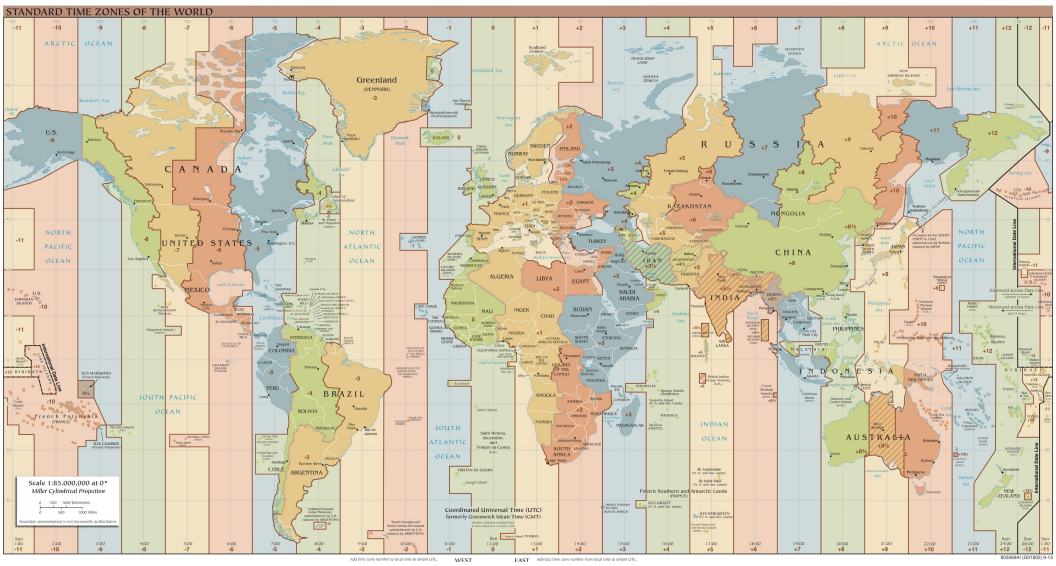

图 1.1: 世界时区划分

日界线

日界线是国际日期变更线的简称，为了统一日期的计时，在东十二区和西十二区附近设立的一条分界线。当从西向东跨过日界线时，日期加一天，从东向西跨过日界线时，日期减一天。[3]

笔记 需要注意的是，日界线并不一定在东十二区和西十二区中间。

1.1.3 ZoneId [4]

ZoneId 是在 Java 1.8 之后引入的时区 Api，用于替代旧的 Api **TimeZone**，可以认为是一个区域的表示。

ZoneId 是一个抽象类，其有两个实现类：

1. **ZoneOffset** 该类为偏移时区，未指定具体区域。
2. **ZoneRegion** 该类表示具体区域的时区。该类为 internal。

这三个类都没有公开构造函数，只能通过他们的静态方法去构建对象。**ZoneId** 的静态方法可以构建出 **ZoneOffset** 或者 **ZoneRegion** 示例，而 **ZoneOffset** 的静态方法只能构建出 **ZoneOffset** 实例。**ZoneRegion** 只能通过 **ZoneId** 的静态方法间接构造。

下面给出了创建 **ZoneId** 的一些代码示例：

```
@Test
fun testZoneIds() {
    // 0时区
    println(ZoneId.of("Z"))
    println(ZoneOffset.UTC)
    assertEquals(ZoneId.of("Z"), ZoneOffset.UTC)

    // UTC时区，需要注意的是ZoneId.of("UTC") != ZoneOffset.UTC
    println(ZoneId.of("UTC"))
    println(ZoneId.of("GMT"))
```

[3]过日界线的计算：https://zhuanlan.zhihu.com/p/250118987

[4]JAVA8 之日期时间时区之 ZoneId[ZoneOffset, ZoneRegion] 笔记：https://blog.csdn.net/kfepiza/article/details/115433132

```
11
12      // 系统默认时区，一般为Asia/Shanghai
13      println(ZoneId.systemDefault())
14
15      // UTC+8时区，下面四个方法是等价的
16      println(ZoneId.of("+8"))
17      println(ZoneId.of("+08:00"))
18      println(ZoneOffset.of("+08:00"))
19      println(ZoneOffset.ofHours(8))
20
21      // Asia/Shanghai时区
22      println(ZoneId.of("Asia/Shanghai"))
23  }
```

我们去研究 **ZoneId** 的代码，发现两个比较重要的抽象成员 **id** 和 **rules**：

```
1   public abstract class ZoneId implements Serializable {
2
3       public abstract String getId();
4
5       public abstract ZoneRules getRules();
6   }
```

其中 **ZoneOffset** 与 **ZoneRegion** 的实现如下：

```
1   public final class ZoneOffset extends ZoneId implements TemporalAccessor,
        TemporalAdjuster, Comparable<ZoneOffset>, Serializable {
2
3       /**
4           Gets the normalized zone offset ID.
5           The ID is minor variation to the standard ISO-8601 formatted string for the
                offset. There are three formats:
6           Z - for UTC (ISO-8601)
7           +hh:mm or -hh:mm - if the seconds are zero (ISO-8601)
8           +hh:mm:ss or -hh:mm:ss - if the seconds are non-zero (not ISO-8601)
9           Returns:
10          the zone offset ID, not null
11       */
12      @Override
13      public String getId() {
14          return id;
15      }
16
17      /**
18          Gets the associated time-zone rules.
19          The rules will always return this offset when queried. The implementation class
                is immutable, thread-safe and serializable.
20          Returns:
21          the rules, not null
```

```
     */
    @Override
    public ZoneRules getRules() {
        return ZoneRules.of(this);
    }
}

final class ZoneRegion extends ZoneId implements Serializable {

    //-----------------------------------------------------------------------
    @Override
    public String getId() {
        return id;
    }

    @Override
    public ZoneRules getRules() {
        // additional query for group provider when null allows for possibility
        // that the provider was updated after the ZoneId was created
        return (rules != null ? rules : ZoneRulesProvider.getRules(id, false));
    }
}
```

我们可以发现 **ZoneId** 对应的具体时间偏移是存储在 **rules** 中的，由于该类的代码非常复杂，就不再深入研究了。同时，由于政策的因素，一个区域所属的时区可能随时发生变更。因此 **ZoneRegion** 的 **rules** 存储在系统资源中。

1.1.4 获取当前时间

我们首先看几个常用的 Api 获取当前时间的例子：

```
@Test
fun testNow() {
    // Old Apis
    val oldDate = Date()
    println(oldDate)
    val oldDate2 = Calendar.getInstance().time
    println(oldDate2)

    // New Apis
    val instant = Instant.now()
    println(instant)

    val localDateTime = LocalDateTime.now()
    println(localDateTime)

    val zonedDateTime = Instant.now().atZone(ZoneId.systemDefault())
    println(zonedDateTime)
```

```
}
```

输入如下：（根据调用的时间会产生不同的结果）

```
Sun Jul 11 08:21:28 CST 2021
Sun Jul 11 08:21:28 CST 2021
2021-07-11T00:21:28.568046200Z
2021-07-11T08:21:28.583665900
2021-07-11T08:21:28.583665900+08:00[Asia/Shanghai]
```

可以发现 **Instant.now()** 获取的是当前的 UTC 时间，然后 **LocalDateTime.now()** 获取的是当前的本地时间。

笔记 CST：区域 Asia/Shanghai 的简写

Date 获取的时间是不带时区信息的。在这五个对象的实例中，仅有 **ZonedDateTime** 带时区信息。

有一定了解的话，我们会发现 **Date** 的 **hours** 等属性是被废弃的，究其原因主要是其不自带时区信息，容易产生混淆。应当使用 **Calandar** 的相关方法进行代替。

正式由于旧 Api 定义不明，使用复杂，精度不高。自 java 1.8 开始。**Instant** 就替代了 **Date**，**LocalDateTime**、**ZonedDateTime** 就替代了 **Calandar**，成为了 java 日期与时间的新的标准。

获得一个确定的时间

加入我们有一个时间 2021/1/1 19:00:00(UTC)，我们怎样通过新或者旧的 Api 去获取这个示例呢。下面给出几种调用的方式：

```
@Test
fun testSpecific() {
    // 首先固定一个时钟 对应本地时间2021/7/12 3:00
    val fixedInstant = Instant.parse("2021-07-11T19:00:00.00Z")
    val fixedClock = Clock.fixed(
        fixedInstant, ZoneId.systemDefault()
    )

    val date = Date(fixedClock.millis())
    println(date)
    val date2 = Calendar.getInstance().apply {
        this.timeInMillis = fixedClock.millis()
    }.time
    println(date2)
    // 特别需要注意：Calendar.getInstance()会获取本地日历系统，设置的时间也是按照本地时间来
        设置的。
    val date3 = Calendar.getInstance().apply {
        this.set(Calendar.YEAR, 2021)
        // 特别需要注意的是，Calendar中Month范围为0~11.
        this.set(Calendar.MONTH, Calendar.JULY)
        this.set(Calendar.DATE, 12)
        this.set(Calendar.HOUR, 3)
        this.set(Calendar.MINUTE, 0)
        this.set(Calendar.SECOND, 0)
        // 这个一定要设置成0
```

```
25          this.set(Calendar.MILLISECOND, 0)
26      }.time
27      println(date3)
28
29      val date4 = Instant.from(fixedInstant)
30      val date5 = Instant.ofEpochMilli(fixedClock.millis())
31      println(date4)
32      println(date5)
33
34      val date6 = LocalDateTime.of(2021,7,12,3,0,0)
35      println(date6)
36
37      val date7 = ZonedDateTime.ofInstant(date4, ZoneId.systemDefault())
38      println(date7)
39  }
```

输入如下：

```
Mon Jul 12 03:00:00 CST 2021
Mon Jul 12 03:00:00 CST 2021
Mon Jul 12 03:00:00 CST 2021
2021-07-11T19:00:00Z
2021-07-11T19:00:00Z
2021-07-12T03:00
2021-07-12T03:00+08:00[Asia/Shanghai]
```

1.1.5 时间从哪来？时钟系统

如何获取当前的时间，就一定需要系统的相关资源，在阅读源码之后，我们会发现，无论是 Old Api 合适 New Api，都是使用到时间戳，即相对于 **EPOCH** 的偏移量。

笔记 在计算机系统中，会有一个时钟，用来记录距离 1970-01-01T00:00:00Z（EPOCH）的毫秒数（或者纳秒数）。

我们先看两者源码的实现：

```
1   public class Date implements java.io.Serializable, Cloneable, Comparable<Date>
2   {
3       public Date() {
4           this(System.currentTimeMillis());
5       }
6   }
7
8   public final class Instant implements Temporal, TemporalAdjuster, Comparable<Instant>,
        Serializable {
9       public static Instant now() {
10          return Clock.systemUTC().instant();
11      }
12  }
13
```

```
14 static final class SystemClock extends Clock implements Serializable {
15     @Override
16     public long millis() {
17         // System.currentTimeMillis() and VM.getNanoTimeAdjustment(offset)
18         // use the same time source - System.currentTimeMillis() simply
19         // limits the resolution to milliseconds.
20         // So we take the faster path and call System.currentTimeMillis()
21         // directly - in order to avoid the performance penalty of
22         // VM.getNanoTimeAdjustment(offset) which is less efficient.
23         return System.currentTimeMillis();
24     }
25     @Override
26     public Instant instant() {
27         // Take a local copy of offset. offset can be updated concurrently
28         // by other threads (even if we haven't made it volatile) so we will
29         // work with a local copy.
30         long localOffset = offset;
31         long adjustment = VM.getNanoTimeAdjustment(localOffset);
32
33         if (adjustment == -1) {
34             // -1 is a sentinel value returned by VM.getNanoTimeAdjustment
35             // when the offset it is given is too far off the current UTC
36             // time. In principle, this should not happen unless the
37             // JVM has run for more than ~136 years (not likely) or
38             // someone is fiddling with the system time, or the offset is
39             // by chance at 1ns in the future (very unlikely).
40             // We can easily recover from all these conditions by bringing
41             // back the offset in range and retry.
42
43             // bring back the offset in range. We use -1024 to make
44             // it more unlikely to hit the 1ns in the future condition.
45             localOffset = System.currentTimeMillis()/1000 - 1024;
46
47             // retry
48             adjustment = VM.getNanoTimeAdjustment(localOffset);
49
50             if (adjustment == -1) {
51                 // Should not happen: we just recomputed a new offset.
52                 // It should have fixed the issue.
53                 throw new InternalError("Offset " + localOffset + " is not in range");
54             } else {
55                 // OK - recovery succeeded. Update the offset for the
56                 // next call...
57                 offset = localOffset;
58             }
59         }
60         return Instant.ofEpochSecond(localOffset, adjustment);
```

```
61      }
62  }
```

我们可以发现，两者通过逐级的调用，都使用到了 **System.currentTimeMillis()**，**VM.getNanoTimeAdjustMent(Long)** 两个方法，两者都是和硬件相关的。

Clock

Clock 是在 java1.8 后引入的新的时钟类，主要用于提供当前的时间信息，实际上就是对系统资源的包装（Facade）。他具有四种实现类 **SystemClock**，**OffsetClock**，**TickClock**，**FixedClock**。

其中 **SystemClock** 是系统时钟类，可以获取当前的时钟信息（系统资源）。而 **FixedClock** 提供一个固定的时钟，常常用于测试。其他的两个实现 **OffsetClock** 和 **TickClock** 都是装饰类，一个添加偏移，一个进行 **truncate** 操作。

1.1.6 Instant、LocalDateTime、ZoneDateTime 之间的关系

1.1.7 时间的属性获取与设置 (Temporal)

1.1.8 时间的加与减

1.1.9 Duration 与 TemporalAmout

1.2 Spring

1.3 Locale

1.4 NIO

第 2 章 Record Logs

2.1 2021 年

2021 年 7 月 10 日创建了1.1Joda-time 时间处理类 (since 1.8)，编写第 1-3 节。

2021 年 7 月 11 日编写1.1Joda-time 时间处理类 (since 1.8) 第 4-5 节。