Java8-时间API

新的API：**java.time**，由5个包组成：

- java.time – 包含值对象的基础包
- java.time.chrono – 提供对不同的日历系统的访问
- java.time.format – 格式化和解析时间和日期
- java.time.temporal – 包括底层框架和扩展特性
- java.time.zone – 包含时区支持的类

1. Instant

1.0 扩展

> Java API defines its own time-scale, the *Java Time-Scale*.
>
> The Java Time-Scale divides each calendar day into exactly 86400 subdivisions, known as seconds. These seconds may differ from the SI second. It closely matches the de facto international civil time scale, the definition of which changes from time to time.

Java时间标准与国际通用时间标准（the international civil time scale， 目前是UTC ）关系：

- 非常接近国际通用时间标准；
- 正午时间与国际通用时间标准一致；
- Java时间标准与国际通用时间标准之间有精确的定义关系
- Java时间标准与**UTC-SLS**（UTC with Smoothed Leap Seconds）等价

1.0.1 Java Time-Scale：

1.0.1.1 Java Time-Scale特点

> - the Java Time-Scale shall closely match the underlying international civil time scale;
> - the Java Time-Scale shall exactly match the international civil time scale at noon each day;
> - the Java Time-Scale shall have a precisely-defined relationship to the international civil time scale.

1.0.2 关于UTC-SLS

> - During the last 1000 seconds of a UTC day with inserted leap second, a UTC-SLS clock slows down to by 0.1% to 0.999 times its normal rate, such that the last 1000 UTC seconds (including the inserted leap second) span exactly the same time as the corresponding 999 "slow seconds" on the UTC-SLS clock.
> - During the last 1000 seconds of a day with deleted leap second, a UTC-SLS clock accelerates by 0.1% to 1.001 times its normal rate, such that the last 1000 UTC seconds (excluding the deleted leap second) span exactly the same time as the corresponding 1001 “fast seconds” on the UTC-SLS clock.
> - At each full hour (and half hour), UTC and UTC-SLS are identical, even right after a leap second.

1.0.3 UT1与UTC

- UT1（类似的其他UT）是观测遥远的星体（星/类星体）而得出的太阳日；
- UTC是根据原子钟进行计时；

1.1 机制

- 包含一个**long**属性，表示 `纪元秒数` （epoch-seconds）

- 纪元时间（epoch） - *1970-01-01T00:00:00Z*

- 包含一个**int**属性，表示每秒中的 `纳秒值` （nanosecond-of-second），范围 `0 ~ 999,999,999`
- Instant的设计初衷是为了便于机器使用，所以，它无法处理那些我们非常容易理解的时间单位：

```
int day = Instant.now().get(ChronoField.DAY_OF_MONTH); // 抛出异常
```

但是，却可以这样

```
int day = Instant.now().get(ChronoField.NANO_OF_SECOND);
```

1.2 比较

Instant是 `value-based` 类，对Instant使用恒等判断，会有难以预期的结果，应当使用 `equals` 方法：

> This is a **value-based** class; use of **identity-sensitive** operations (*including reference equality (==), identity hash code, or synchronization*) on instances of Instant may have unpredictable results and should be avoided. The `equals` method should be used for comparisons.

1.3 API

- **ofEpochSecond(long epochSecond, long nanoAdjustment)**
  - **nanoAdjustment** - 当 `nanoAdjustment` 大于等于 `1,000,000,000` 时，先将 `nanoAdjustment` 中整秒（nanoAdjustment/10^9）部分添加到 `epochSecond` 参数上， `余数` 部分再设置成 `纳秒值`

    例如：下面的代码创建同样的Instant对象

    ```
    Instant.ofEpochSecond(3);
    Instant.ofEpochSecond(3, 0);
    Instant.ofEpochSecond(2, 1_000_000_000);
    Instant.ofEpochSecond(4, -1_000_000_000);
    ```

- **ofEpochMilli(long epochMilli)**
  - **epochMilli** - 类似上一个接口中的nanoAdjustment参数，也是会将整秒（epochMilli/10^3）部分设置到epochSecond上（此处epochSecond默认为0）

2. Temporal

Temporal是框架级的最主要的接口之一；定义着许多时间相关对象的读写权限

> Framework-level interface defining read-write access to a temporal object, such as a date, time, offset or some combination of these.

2.1 接口方法

- get/getLong - 获取某个单位的对应的时间值（继承自TemporalAccessor）
- with - 设置某个单位的时间值
- minus - 时间减法
- plus - 时间加法
- isSupported - 是否支持某个时间单位（比如：年、月等）
  - 如果不支持某个时间单位，却没有进行检查而直接使用（比如：get方法），会抛出 `UnsupportedTemporalTypeException` 异常。

3. Duration

Duration的toString结果格式：

- **±P**n**DT**n**H**n**M**n.n**S** - 代表：±(n天n小时n分n.n秒) —— 4个部分
  - *(基于ISO-8601标准)*
  - **T**必须在时间块前面出现（H/M/S之前出现），如果**T**存在，那么**T**后面必须有东西（H或M或S）
  - n - 可以带正负号
  - **D/H/M**前面的n必须可以转换成**long**型
  - **S**前面的n包含2部分：

    格式： `秒数` `.|,` `纳秒数`

    纳秒数 - 范围不超过9位数

    > The number of seconds must parse to an long with optional fraction. The decimal point may be either a dot or a comma.

4. TimeZone

TimeZone#getDefault()

- 如果有默认TimeZone并且可用，则会返回默认TimeZone的clone版本
- 否则：
  - 使用环境属性 `user.timezone` 的值作为时区ID
  - 探测系统平台的时区ID
  - 使用GMT

5. ZoneOffset

> ZoneId的子类，ZoneOffset，代表了这种从伦敦格林威治零度子午线开始的时间偏移。

这个值是会变化的：主要原因是有些国家实行时令机制

> 例：大家想感受一下冬令时的话，在系统->通用->日期与时间里 关闭自动设置，然后打开时区搜索 哈瓦那(古巴的一个时区)，然后将时间调整到2015年11月1日，00:59。等一分钟时间会自动跳到00:00

在中国：

> 在古代中国，冬至白天45刻，随后每九天加一刻，至夏至65刻，然后每九天减一刻。钦天监每九天换一次漏尺。

科普：

> UTC 是 Coordinated Universal Time 的缩写，译为中文为“世界标准时间”，直译的话，可译为“协调通用时间”或“协调世界时间”。目前来说也就是指 GMT 时间。为什么说目前就是指 GMT 时间呢？因为本初子午线（子午线即经线，本初子午线即 0 度经线）其实穿过的是沙特阿拉伯西边的麦加，而不是英国的格林威治。当时英国皇家学会暂时确定格林威治为本初子午线的穿过点，加之英国正是兴旺发达时期，全世界就将错就错，用到现在。说不定哪天改为麦加时间为标准时间也不是没有可能。所以我们一般使用 UTC，而不是 GMT。

代码考虑：

如果你想证明，在夏令时结束那天的重叠时段，你有考虑过什么情况会发生，你可以用这两个专门处理重叠时段的方法之一：

```
zdt = zdt.withEarlierOffsetAtOverlap();
zdt = zdt.withLaterOffsetAtOverlap();
```

6. LocalDate

`LocalDate` 是不可变类型，每次操作都会产生一个新的实例，而原有实例不收任何影响；

`LocalDate` 是**值类型**（查看下文中"扩展"部分），所以应该避免使用 `==` 来判断2个LocalDate对象；

例如：

```
/* 此处的date并不会变化，计算后的日期结果是一个新的日期对象 */
LocalDate ld = date.withYear(2011);
```

7. TemporalAdjuster

时间调整器（修改器），提供一个预装包的、能操纵日期的功能，比如，根据月份的最后一天获取日期的对象

```
import static java.time.DayOfWeek.*  // static
import static java.time.temporal.TemporalAdjusters.*

LocalDate date = LocalDate.of(2014, Month.JUNE, 10);
date = date.with(lastDayOfMonth());
date = date.with(nextOrSame(WEDNESDAY));
```

7.1 Lambda

由于TemporalAdjuster只有一个 `adjustInto` 方法，这使得TemporalAdjuster可以使用Lambda表达式；

- 建议使用**TemporalAdjusters#ofDateAdjuster**进行Lambda编程；

8. Chronology

Chronology接口，是其他历法的主要入口点，它允许通过所属的语言环境查找对应的历法系统;

- Java 8支持额外的4个历法系统：泰国佛教历，中华民国历(**MinguoDate**)，日本历（沿袭中国古代帝位纪年），伊斯兰历；如有需要，应用程序也可以实现自己的历法系统；

9. LocalTime

LocalTime是值类型，且跟日期和时区没有关联。当我们对时间进行加减操作时，以午夜基准，24小时一个周期。因此，20:00加上6小时，结果就是02:00。

10. LocalDateTime

LocalDateTime的其他方法跟**LocalDate**和**LocalTime**相似。这种相似的方法模式非常有利于API的学习。下面总结了用到的方法前缀：

- of： 静态工厂方法，从组成部分中创建实例
- from： 静态工厂方法，尝试从相似对象中提取实例。from()方法没有of()方法类型安全
- now： 静态工厂方法，用当前时间创建实例
- parse： 静态工厂方法，总字符串解析得到对象实例
- get： 获取时间日期对象的部分状态
- is： 检查关于时间日期对象的描述是否正确
- with： 返回一个部分状态改变了的时间日期对象拷贝
- plus： 返回一个时间增加了的、时间日期对象拷贝
- minus： 返回一个时间减少了的、时间日期对象拷贝
- to： 把当前时间日期对象转换成另外一个，可能会损失部分状态

- at： 用当前时间日期对象组合另外一个，创建一个更大或更复杂的时间日期对象
- format： 提供格式化时间日期对象的能力

10.1 LocalDateTime、Localdate、LocalTime转换

10.1.1 Localdate或LocalTime → LocalDateTime

```
LocalDateTime dt4 = date.atTime(time);
LocalDateTime dt5 = time.atDate(date);
```

10.1.2 LocalDateTime →Localdate或LocalTime

```
LocalDate ld = localDateTime.toLocalDate();
LocalTime lt = localDateTime.toLocalTime();
```

11. ZoneIdl

在**java.time**之前，我们用**TimeZone**表示时区，而现在，用**ZoneId**。区别：

- **ZoneId**是不可变的，它里面保存时区缩写的静态变量也是不可变的；
- 实际的规则集在**ZoneRules**里，不在**ZoneId**中，通过**getRules()**方法可以获得；

11.1 TimeZone转ZoneId

```
ZoneId zoneId = TimeZone.getDefault().toZoneId();
```

11.2 ZoneId与其他时间类

- ZoneId可以与LocalDate、LocalDateTime或者是Instant对象整合起来，构造为一个**ZonedDateTime**实例

```
LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
ZonedDateTime zdt1 = date.atStartOfDay(romeZone);

LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
ZonedDateTime zdt2 = dateTime.atZone(romeZone);

Instant instant = Instant.now();
ZonedDateTime zdt3 = instant.atZone(romeZone);
```

- 通过ZoneId，你还可以将LocalDateTime转换为Instant

```
LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
Instant instantFromDateTime = dateTime.toInstant(romeZone);
```

- 可以通过反向的方式得到LocalDateTime对象

```
Instant instant = Instant.now();
LocalDateTime timeFromInstant = LocalDateTime.ofInstant(instant, romeZone);
```

12. OffsetDateTime

利用当前时间和伦敦格林尼治子午线时间的差异来表示本地时间，如以纽约为例：

```
ZoneOffset newYorkOffset = ZoneOffset.of("-05:00");

LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
OffsetDateTime dateTimeInNewYork = OffsetDateTime.of(dateTime, newYorkOffset);
// 结果：2014-03-18T13:45-05:00
```

13. ZoneDateTime

JDK便利：

> 作为一个开发者，如果不用去处理时区和它带来的复杂性，将会是非常棒的。**java.time**开发包尽最大努力的帮助你那样做。只要有可能，尽量使用**LocalDate**，**LocalTime**，**LocalDate**和**Instant**。当你不能回避时区时，**ZonedDateTime**可以满足你的需求。

14. 时间长度

- **Duration**表示以秒和纳秒为基准的时长。例如，“23.6秒”；
- **Period**表示以年、月、日衡量的时长。例如，“3年2个月零6天”；

15. 格式化

DateTimeFormatter类，以及它的辅助类**DateTimeFormatterBuilder**；

示例：

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("dd/MM/uuuu");
LocalDate date = LocalDate.parse("24/06/2014", f);
String str = date.format(f);
```

15.1 DateFormat和DateTimeFormatter

> 和老的java.util.DateFormat相比较，所有的DateTimeFormatter实例都是线程安全的。所以，你能够以单例模式创建格式器实例，就像DateTimeFormatter所定义的那些常量，并能在多个线程间共享这些实例。

15.2 编程式创建DateTimeFormatter

```
DateTimeFormatter italianFormatter = new DateTimeFormatterBuilder()
  .appendText(ChronoField.DAY_OF_MONTH)
  .appendLiteral(". ")
  .appendText(ChronoField.MONTH_OF_YEAR)
  .appendLiteral(" ")
  .appendText(ChronoField.YEAR)
  .parseCaseInsensitive()
  .toFormatter(Locale.ITALIAN);
```

16. Period

16.1 Period、Duration

> 由于LocalDateTime和Instant是为不同的目的而设计的，一个是为了便于人阅读使用，另一个是为了便于机器处理，所以你不能将二者混用。如果你试图在这两类对象之间创建duration，会触发一个DateTimeException异常

> 如果你需要以年、月或者日的方式对多个时间单位建模，可以使用Period类

N. 扩展

N.1 UTC与UTC-SLS区别：

- 除了最后1000秒（包含插入的闰秒），之前的秒数UTC和UTC-SLS没有区别；
- UTC-SLS是如何消化掉闰秒（Leap Second）的：
  - 当插入闰秒的时候，最后1000 UTC秒（包括插入的闰秒）等分成UTC-SLS的999秒

    > During the last 1000 seconds of a UTC day with inserted leap second, a UTC-SLS clock slows down to by 0.1% to 0.999 times its normal rate, such that the last 1000 UTC seconds (including the inserted leap second) span exactly the same time as the corresponding 999 "slow seconds" on the UTC-SLS clock.

  - 当删除闰秒的时候，最后1000 UTC秒（不包括删除的闰秒）等分成UTC-SLS的1001秒

    > During the last 1000 seconds of a day with deleted leap second, a UTC-SLS clock accelerates by 0.1% to 1.001 times its normal rate, such that the last 1000 UTC seconds (excluding the deleted leap second) span exactly the same time as the corresponding 1001 “fast seconds” on the UTC-SLS clock.

- 在每个整小时（和半小时），UTC和UTC-SLS是相同的;

Math

- **Math.multiplyExact(a, b)**
  - 结果： a * b
- 判断2个数是不是都为0
  - numberA | numberB == 0
- ISO-8601中关于week的定义：

  > The first day-of-week. For example, the ISO-8601 standard considers Monday to be the first day-of-week. The minimal number of days in the first week. For example, the ISO-8601 standard counts the first week as needing at least 4 days.

  - 星期一是一周的第一天
  - 在判断当前星期是否是更大度量（比如：年）的第一周时，当前周必须要有4天落在更大度量范围内；

值类型

值类型：2个实例，只要内容相同，就该是可以相互替换的，是不是同一个实例，并不重要

如：**String**类就是一个标准的值类型的例子，只要字面值一样，我们就认为它们相等，而不关心它们是不是同一个**String**对象的不同引用

- 但是String类型有些特别：

```
String s1 = new String("hongxing");
String s2 = new String("hongxing");
System.out.println(s1.hashCode() == s2.hashCode()); // 结果：true
```

> 对于值类型，想要更多了解的同学，可以参考文章，VALJOs：在Java中，它对值类型，定义了严格的规则集合，包括不可变性、工厂方法和良好定义的**equals()**，**hashCode()**，**toString()**，**compareTo()**方法。

情话

这是机器中的Forever

```
FOREVER("Forever", Duration.ofSeconds(Long.MAX_VALUE, 999_999_999)); // 大约2922亿年，远可以描述到宇宙的形成了
```

却不敌我对你的爱的万分之一