Java 8 - LocalDate/LocalDateTime

理解时间和日期库需要理解如下问题:

- Java8之前的Date有哪些槽点?
- Java8之前使用哪些常用的第三方时间库?
- Java8关于时间和日期有哪些类和方法，变比Java8之前它的特点是什么?
- 其它语言时间库?

Java8之前的Date有哪些槽点

> Tiago Fernandez做过一次投票，选举最烂的JAVA API，排第一的EJB2.X，第二的就是日期API。

槽点一

最开始的时候，Date既要承载日期信息，又要做日期之间的转换，还要做不同日期格式的显示，职责较繁杂(不懂单一职责，你妈妈知道吗? 纯属恶搞~哈哈)

后来从JDK 1.1 开始，这三项职责分开了:

```
使用Calendar类实现日期和时间字段之间转换;
使用DateFormat类来格式化和分析日期字符串;
而Date只用来承载日期和时间信息。
```

原有Date中的相应方法已废弃。不过，无论是Date，还是Calendar，都用着太不方便了，这是API没有设计好的地方。

槽点二

坑爹的year和month

```
Date date = new Date(2012,1,1);
System.out.println(date);
输出Thu Feb 01 00:00:00 CST 3912
```

观察输出结果，year是2012+1900，而month，月份参数我不是给了1吗? 怎么输出二月(Feb)了?

应该曾有人告诉你，如果你要设置日期，应该使用 java.util.Calendar，像这样…

```
Calendar calendar = Calendar.getInstance();
calendar.set(2013, 8, 2);
```

这样写又不对了，calendar的month也是从0开始的，表达8月份应该用7这个数字，要么就干脆用枚举

```
calendar.set(2013, Calendar.AUGUST, 2);
```

注意上面的代码，Calendar年份的传值不需要减去1900(当然月份的定义和Date还是一样)，这种不一致真是让人抓狂!

有些人可能知道，Calendar相关的API是IBM捐出去的，所以才导致不一致。

槽点三

java.util.Date与java.util.Calendar中的所有属性都是可变的

下面的代码，计算两个日期之间的天数....

```
public static void main(String[] args) {
    Calendar birth = Calendar.getInstance();
    birth.set(1975, Calendar.MAY, 26);
    Calendar now = Calendar.getInstance();
    System.out.println(daysBetween(birth, now));
    System.out.println(daysBetween(birth, now)); // 显示 0?
 }

public static long daysBetween(Calendar begin, Calendar end) {
    long daysBetween = 0;
    while(begin.before(end)) {
        begin.add(Calendar.DAY_OF_MONTH, 1);
        daysBetween++;
    }
    return daysBetween;
}
```

daysBetween有点问题，如果连续计算两个Date实例的话，第二次会取得0，因为Calendar状态是可变的，考虑到重复计算的场合，最好复制一个新的Calendar

```
public static long daysBetween(Calendar begin, Calendar end) {
    Calendar calendar = (Calendar) begin.clone(); // 复制
    long daysBetween = 0;
    while(calendar.before(end)) {
        calendar.add(Calendar.DAY_OF_MONTH, 1);
        daysBetween++;
    }
    return daysBetween;
}
```

槽点四

SimpleDateTimeFormat是非线程安全的。

Java8时间和日期

类概览

Java 8仍然延用了ISO的日历体系，并且与它的前辈们不同，java.time包中的类是不可变且线程安全的。新的时间及日期API位于java.time包中，下面是里面的一些关键的类:

- Instant——它代表的是时间戳
- LocalDate——不包含具体时间的日期，比如2014-01-14。它可以用来存储生日，周年纪念日，入职日期等。
- LocalTime——它代表的是不含日期的时间

- LocalDateTime——它包含了日期及时间，不过还是没有偏移信息或者说时区。
- ZonedDateTime——这是一个包含时区的完整的日期时间，偏移量是以UTC/格林威治时间为基准的。

新的库还增加了ZoneOffset及Zoned，可以为时区提供更好的支持。有了新的DateTimeFormatter之后日期的解析及格式化也变得焕然一新了。

方法概览

该包的API提供了大量相关的方法，这些方法一般有一致的方法前缀:

- of: 静态工厂方法。
- parse: 静态工厂方法，关注于解析。
- get: 获取某些东西的值。
- is: 检查某些东西的是否是true。
- with: 不可变的setter等价物。
- plus: 加一些量到某个对象。
- minus: 从某个对象减去一些量。
- to: 转换到另一个类型。
- at: 把这个对象与另一个对象组合起来，例如: date.atTime(time)。

一些例子

```
public class TimeIntroduction {
    public static void testClock() throws InterruptedException {
        //时钟提供给我们用于访问某个特定 时区的 瞬时时间、日期 和 时间的。
        Clock c1 = Clock.systemUTC(); //系统默认UTC时钟(当前瞬时时间 System.currentTimeMillis())
        System.out.println(c1.millis()); //每次调用将返回当前瞬时时间(UTC)
        Clock c2 = Clock.systemDefaultZone(); //系统默认时区时钟(当前瞬时时间)
        Clock c31 = Clock.system(ZoneId.of("Europe/Paris")); //巴黎时区
        System.out.println(c31.millis()); //每次调用将返回当前瞬时时间(UTC)
        Clock c32 = Clock.system(ZoneId.of("Asia/Shanghai"));//上海时区
        System.out.println(c32.millis());//每次调用将返回当前瞬时时间(UTC)
        Clock c4 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Shanghai"));//固定上海时区时钟
        System.out.println(c4.millis());
        Thread.sleep(1000);
        System.out.println(c4.millis()); //不变 即时钟时钟在那一个点不动
        Clock c5 = Clock.offset(c1, Duration.ofSeconds(2)); //相对于系统默认时钟两秒的时钟
        System.out.println(c1.millis());
        System.out.println(c5.millis());
    }
    public static void testInstant() {
        //瞬时时间 相当于以前的System.currentTimeMillis()
        Instant instant1 = Instant.now();
        System.out.println(instant1.getEpochSecond());//精确到秒 得到相对于1970-01-01 00:00:00
UTC的一个时间
        System.out.println(instant1.toEpochMilli()); //精确到毫秒
        Clock clock1 = Clock.systemUTC(); //获取系统UTC默认时钟
        Instant instant2 = Instant.now(clock1);//得到时钟的瞬时时间
        System.out.println(instant2.toEpochMilli());
        Clock clock2 = Clock.fixed(instant1, ZoneId.systemDefault()); //固定瞬时时间时钟
        Instant instant3 = Instant.now(clock2);//得到时钟的瞬时时间
        System.out.println(instant3.toEpochMilli());//equals instant1
    }
    public static void testLocalDateTime() {
```

```
        //使用默认时区时钟瞬时时间创建 Clock.systemDefaultZone() -->即相对于 ZoneId.systemDefault()
默认时区
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);
        //自定义时区
        LocalDateTime now2 = LocalDateTime.now(ZoneId.of("Europe/Paris"));
        System.out.println(now2);//会以相应的时区显示日期
        //自定义时钟
        Clock clock = Clock.system(ZoneId.of("Asia/Dhaka"));
        LocalDateTime now3 = LocalDateTime.now(clock);
        System.out.println(now3);//会以相应的时区显示日期
        //不需要写什么相对时间 如java.util.Date 年是相对于1900 月是从0开始
        //2013-12-31 23:59

        LocalDateTime d1 = LocalDateTime.of(2013, 12, 31, 23, 59);
        //年月日 时分秒 纳秒
        LocalDateTime d2 = LocalDateTime.of(2013, 12, 31, 23, 59, 59, 11);
        //使用瞬时时间 + 时区
        Instant instant = Instant.now();
        LocalDateTime d3 = LocalDateTime.ofInstant(Instant.now(), ZoneId.systemDefault());
        System.out.println(d3);

        //解析String--->LocalDateTime
        LocalDateTime d4 = LocalDateTime.parse("2013-12-31T23:59");
        System.out.println(d4);
        LocalDateTime d5 = LocalDateTime.parse("2013-12-31T23:59:59.999");//999毫秒 等价于
999000000纳秒
        System.out.println(d5);

        //使用DateTimeFormatter API 解析 和 格式化
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        LocalDateTime d6 = LocalDateTime.parse("2013/12/31 23:59:59", formatter);
        System.out.println(formatter.format(d6));

        //时间获取
        System.out.println(d6.getYear());
        System.out.println(d6.getMonth());
        System.out.println(d6.getDayOfYear());
        System.out.println(d6.getDayOfMonth());
        System.out.println(d6.getDayOfWeek());
        System.out.println(d6.getHour());
        System.out.println(d6.getMinute());
        System.out.println(d6.getSecond());
        System.out.println(d6.getNano());

        //时间增减
        LocalDateTime d7 = d6.minusDays(1);
        LocalDateTime d8 = d7.plus(1, IsoFields.QUARTER_YEARS);
        //LocalDate 即年月日 无时分秒
        //LocalTime即时分秒 无年月日
        //API和LocalDateTime类似就不演示了

        // 两个日期是否相等
        System.out.println(d1.equals(d2));

        // MonthDay - 用来检查生日
        LocalDate dateOfBirth = LocalDate.of(2010, 01, 14);
        MonthDay birthday = MonthDay.of(dateOfBirth.getMonth(), dateOfBirth.getDayOfMonth());
        MonthDay currentMonthDay = MonthDay.from(today);
        System.out.println(currentMonthDay.equals(birthday));
```

```
        // YearMonth - 用来检查信用卡过期
        YearMonth currentYearMonth = YearMonth.now(); System.out.printf("Days in month year %s:
%d%n", currentYearMonth, currentYearMonth.lengthOfMonth());
        YearMonth creditCardExpiry = YearMonth.of(2018, Month.FEBRUARY);
        System.out.printf("Your credit card expires on %s %n", creditCardExpiry);

        // 判断闰年 - LocalDate类有一个isLeapYear()的方法
        System.out.println(dateOfBirth.isLeapYear());
    }
    public static void testZonedDateTime() {
        //即带有时区的date-time 存储纳秒、时区和时差(避免与本地date-time歧义)。
        //API和LocalDateTime类似，只是多了时差(如2013-12-20T10:35:50.711+08:00[Asia/Shanghai])
        ZonedDateTime now = ZonedDateTime.now();
        System.out.println(now);
        ZonedDateTime now2 = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
        System.out.println(now2);
        //其他的用法也是类似的 就不介绍了
        ZonedDateTime z1 = ZonedDateTime.parse("2013-12-31T23:59:59Z[Europe/Paris]");
        System.out.println(z1);
    }
    public static void testDuration() {
        //表示两个瞬时时间的时间段
        Duration d1 = Duration.between(Instant.ofEpochMilli(System.currentTimeMillis() -
12323123), Instant.now());
        //得到相应的时差
        System.out.println(d1.toDays());
        System.out.println(d1.toHours());
        System.out.println(d1.toMinutes());
        System.out.println(d1.toMillis());
        System.out.println(d1.toNanos());
        //1天时差 类似的还有如ofHours()
        Duration d2 = Duration.ofDays(1);
        System.out.println(d2.toDays());
    }
    public static void testChronology() {
        //提供对java.util.Calendar的替换，提供对年历系统的支持
        Chronology c = HijrahChronology.INSTANCE;
        ChronoLocalDateTime d = c.localDateTime(LocalDateTime.now());
        System.out.println(d);
    }
    /**
     * 新旧日期转换
     */
    public static void testNewOldDateConversion(){
        Instant instant=new Date().toInstant();
        Date date=Date.from(instant);
        System.out.println(instant);
        System.out.println(date);
    }
    public static void main(String[] args) throws InterruptedException {
        testClock();
        testInstant();
        testLocalDateTime();
        testZonedDateTime();
        testDuration();
        testChronology();
        testNewOldDateConversion();
    }
}
```

其它语言时间

日期与时间处理API，在各种语言中，可能都只是个不起眼的API，如果你没有较复杂的时间处理需求，可能只是利用日期与时间处理API取得系统时间，简单做些显示罢了，然而如果认真看待日期与时间，其复杂程度可能会远超过你的想象，天文、地理、历史、政治、文化等因素，都会影响到你对时间的处理。所以在处理时间上，最好选用JSR310(如果你用java8的话就实现310了)，或者Joda-Time。

不止是java面临时间处理的尴尬，其他语言同样也遇到过类似的问题，比如

Arrow: Python 中更好的日期与时间处理库

Moment.js: JavaScript 中的日期库

Noda-Time: .NET 阵营的 Joda-Time 的复制

总结

看完了这些例子后，对Java 8这套新的时间日期API有了一定的了解了。回顾下关于这个新的API的一些关键的要素。

- 它提供了javax.time.ZoneId用来处理时区。
- 它提供了LocalDate与LocalTime类 Java 8中新的时间与日期API中的所有类都是不可变且线程安全的，这与之前的Date与Calendar API中的恰好相反，那里面像java.util.Date以及SimpleDateFormat这些关键的类都不是线程安全的。
- 新的时间与日期API中很重要的一点是它定义清楚了基本的时间与日期的概念，比方说，瞬时时间，持续时间，日期，时间，时区以及时间段。它们都是基于ISO日历体系的。
- 每个Java开发人员都应该至少了解这套新的API中的这五个类: Instant 它代表的是时间戳，比如2014-01-14T02:20:13.592Z，这可以从java.time.Clock类中获取，像这样：Instant current = Clock.system(ZoneId.of(“Asia/Tokyo”)).instant(); LocalDate 它表示的是不带时间的日期，比如2014-01-14。它可以用来存储生日，周年纪念日，入职日期等。 LocalTime – 它表示的是不带日期的时间 LocalDateTime – 它包含了时间与日期，不过没有带时区的偏移量 ZonedDateTime – 这是一个带时区的完整时间，它根据UTC/格林威治时间来进行时区调整
- 这个库的主包是java.time，里面包含了代表日期，时间，瞬时以及持续时间的类。它有两个子package，一个是java.time.foramt，这个是什么用途就很明显了，还有一个是java.time.temporal，它能从更低层面对各个字段进行访问。
- 时区指的是地球上共享同一标准时间的地区。每个时区都有一个唯一标识符，同时还有一个地区/城市(Asia/Tokyo)的格式以及从格林威治时间开始的一个偏移时间。比如说，东京的偏移时间就是+09:00。OffsetDateTime类实际上包含了LocalDateTime与ZoneOffset。它用来表示一个包含格林威治时间偏移量(+/-小时:分，比如+06:00或者 -08: 00)的完整的日期(年月日)及时间(时分秒，纳秒)。 DateTimeFormatter类用于在Java中进行日期的格式化与解析。与SimpleDateFormat不同，它是不可变且线程安全的，如果需要的话，可以赋值给一个静态变量。DateTimeFormatter类提供了许多预定义的格式器，你也可以自定义自己想要的格式。当然了，根据约定，它还有一个parse()方法是用于将字符串转换成日期的，如果转换期间出现任何错误，它会抛出DateTimeParseException异常。类似的，DateFormatter类也有一个用于格式化日期的format()方法，它出错的话则会抛出DateTimeException异常。
- 再说一句，“MMM d yyyy”与“MMm dd yyyy”这两个日期格式也略有不同，前者能识别出”Jan 2 2014″与”Jan 14 2014″这两个串，而后者如果传进来的是”Jan 2 2014″则会报错，因为它期望月份处传进来的是两个字符。为了解决这个问题，在天为个位数的情况下，你得在前面补0，比如”Jan 2 2014″应该改为”Jan 02 2014″。