

## 16 | 用好Java 8的日期时间类，少踩一些“老三样”的坑

2020-04-16 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



**讲述：王少泽**

时长 21:28 大小 19.66M



你好，我是朱晔。今天，我来和你说说恼人的时间错乱问题。

在 Java 8 之前，我们处理日期时间需求时，使用 Date、Calendar 和 SimpleDateFormat，来声明时间戳、使用日历处理日期和格式化解析日期时间。但是，这些类的 API 的缺点比较明显，比如可读性差、易用性差、使用起来冗余繁琐，还有线程安全问题。

因此，Java 8 推出了新的日期时间类。每一个类功能明确清晰、类之间协作简单、API 定义清晰不踩坑，API 功能强大无需借助外部工具类即可完成操作，并且线程安全。



但是，Java 8 刚推出的时候，诸如序列化、数据访问等类库都还不支持 Java 8 的日期时间类型，需要在新老类中来回转换。比如，在业务逻辑层使用 `LocalDateTime`，存入数据库或者返回前端的时候还要切换回 `Date`。因此，很多同学还是选择使用老的日期时间类。


现在几年时间过去了，几乎所有的类库都支持了新日期时间类型，使用起来也不会有来回切换等问题了。但，很多代码中因为还是用的遗留的日期时间类，因此出现了很多时间错乱的错误实践。比如，试图通过随意修改时区，使读取到的数据匹配当前时钟；再比如，试图直接对读取到的数据做加、减几个小时的操作，来“修正数据”。

今天，我就重点与你分析下时间错乱问题背后的原因，看看使用遗留的日期时间类，来处理日期时间初始化、格式化、解析、计算等可能会遇到的问题，以及如何使用新日期时间类来解决。

## 初始化日期时间


我们先从日期时间的初始化看起。如果要初始化一个 2019 年 12 月 31 日 11 点 12 分 13 秒这样的时间，可以使用下面的两行代码吗？

```
1 Date date = new Date(2019, 12, 31, 11, 12, 13);
2 System.out.println(date);
```

 复制代码

可以看到，输出的时间是 3029 年 1 月 31 日 11 点 12 分 13 秒：

```
1 Sat Jan 31 11:12:13 CST 3920
```

 复制代码

相信看到这里，你会说这是新手才会犯的低级错误：年应该是和 1900 的差值，月应该是从 0 到 11 而不是从 1 到 12。

```
1 Date date = new Date(2019 - 1900, 11, 31, 11, 12, 13);
```

 复制代码

你说的没错，但更重要的问题是，当有国际化需求时，需要使用 Calendar 类来初始化时间。

使用 Calendar 改造之后，初始化时年参数直接使用当前年即可，不过月需要注意是从 0 到 11。当然，你也可以直接使用 Calendar.DECEMBER 来初始化月份，更不容易犯错。为了说明时区的问题，我分别使用当前时区和纽约时区初始化了两次相同的日期：

 复制代码

```
1 Calendar calendar = Calendar.getInstance();
2 calendar.set(2019, 11, 31, 11, 12, 13);
3 System.out.println(calendar.getTime());
4 Calendar calendar2 = Calendar.getInstance(TimeZone.getTimeZone("America/New_Yo
5 calendar2.set(2019, Calendar.DECEMBER, 31, 11, 12, 13);
6 System.out.println(calendar2.getTime());
```

输出显示了两个时间，说明时区产生了作用。但，我们更习惯年 / 月 / 日 时: 分: 秒这样的日期时间格式，对现在输出的日期格式还不满意：

 复制代码

```
1 Tue Dec 31 11:12:13 CST 2019
2 Wed Jan 01 00:12:13 CST 2020
```

那，时区的问题是怎么回事，又怎么格式化需要输出的日期时间呢？接下来，我就与你逐一分析下这两个问题。


## “恼人”的时区问题

我们知道，全球有 24 个时区，同一个时刻不同时区（比如中国上海和美国纽约）的时间是不一样的。对于需要全球化的项目，如果初始化时间时没有提供时区，那就不是一个真正意义上的时间，只能认为是我看到的当前时间的一个表示。

关于 Date 类，我们要有两点认识：

一是，Date 并无时区问题，世界上任何一台计算机使用 new Date() 初始化得到的时间都一样。因为，Date 中保存的是 UTC 时间，UTC 是以原子钟为基础的统一时间，不以太阳参照计时，并无时区划分。

二是，Date 中保存的是一个时间戳，代表的是从 1970 年 1 月 1 日 0 点（Epoch 时间）到现在的毫秒数。尝试输出 Date(0)：

 复制代码

```
1 System.out.println(new Date(0));  
2 System.out.println(TimeZone.getDefault().getID() + ":" + TimeZone.getDefault().getOffset(0));
```

我得到的是 1970 年 1 月 1 日 8 点。因为我机器当前的时区是中国上海，相比 UTC 时差 +8 小时：

 复制代码

```
1 Thu Jan 01 08:00:00 CST 1970  
2 Asia/Shanghai:8
```

对于国际化（世界各国的人都在使用）的项目，处理好时间和时区问题首先就是要正确保存日期时间。这里有两种保存方式：

方式一，以 UTC 保存，保存的时间没有时区属性，是不涉及时区时间差问题的世界统一时间。我们通常说的时间戳，或 Java 中的 Date 类就是用的这种方式，这也是推荐的方式。

方式二，以字面量保存，比如年 / 月 / 日 时: 分: 秒，一定要同时保存时区信息。只有有了时区信息，我们才能知道这个字面量时间真正的时间点，否则它只是一个给人看的时间表示，只在当前时区有意义。Calendar 是有时区概念的，所以我们通过不同的时区初始化 Calendar，得到了不同的时间。

正确保存日期时间之后，就是正确展示，即我们要使用正确的时区，把时间点展示为符合当前时区的时间表示。到这里，我们就能理解为什么会有所谓的“时间错乱”问题了。接下来，我再通过实际案例分析一下，从字面量解析成时间和从时间格式化为字面量这两类问题。

**第一类是**，对于同一个时间表示，比如 2020-01-02 22:00:00，不同时区的人转换成 Date 会得到不同的时间（时间戳）：

[复制代码](#)

```
1 String stringDate = "2020-01-02 22:00:00";
2 SimpleDateFormat inputFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
3 //默认时区解析时间表示
4 Date date1 = inputFormat.parse(stringDate);
5 System.out.println(date1 + ":" + date1.getTime());
6 //纽约时区解析时间表示
7 inputFormat.setTimeZone(TimeZone.getTimeZone("America/New_York"));
8 Date date2 = inputFormat.parse(stringDate);
9 System.out.println(date2 + ":" + date2.getTime());
```

可以看到，把 2020-01-02 22:00:00 这样的时间表示，对于当前的上海时区和纽约时区，转化为 UTC 时间戳是不同的时间：

[复制代码](#)

```
1 Thu Jan 02 22:00:00 CST 2020:1577973600000
2 Fri Jan 03 11:00:00 CST 2020:1578020400000
```

这正是 UTC 的意义，并不是时间错乱。对于同一个本地时间的表示，不同时区的人解析得到的 UTC 时间一定是不同的，反过来不同的本地时间可能对应同一个 UTC。

**第二类问题是**，格式化后出现的错乱，即同一个 Date，在不同的时区下格式化得到不同的时间表示。比如，在我的当前时区和纽约时区格式化 2020-01-02 22:00:00：

[复制代码](#)

```
1 String stringDate = "2020-01-02 22:00:00";
2 SimpleDateFormat inputFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
3 //同一Date
4 Date date = inputFormat.parse(stringDate);
5 //默认时区格式化输出：
6 System.out.println(new SimpleDateFormat("[yyyy-MM-dd HH:mm:ss Z]").format(date));
7 //纽约时区格式化输出
8 TimeZone.setDefault(TimeZone.getTimeZone("America/New_York"));
9 System.out.println(new SimpleDateFormat("[yyyy-MM-dd HH:mm:ss Z]").format(date));
```

输出如下，我当前时区的 Offset（时差）是 +8 小时，对于 -5 小时的纽约，晚上 10 点对应早上 9 点：

```
1 [2020-01-02 22:00:00 +0800]
2 [2020-01-02 09:00:00 -0500]
```

[复制代码](#)

因此，有些时候数据库中相同的时间，由于服务器的时区设置不同，读取到的时间表示不同。这，不是时间错乱，正是时区发挥了作用，因为 UTC 时间需要根据当前时区解析为正确的本地时间。

所以，**要正确处理时区，在于存进去和读出来两方面**：存的时候，需要使用正确的当前时区来保存，这样 UTC 时间才会正确；读的时候，也只有正确设置本地时区，才能把 UTC 时间转换为正确的当地时间。

Java 8 推出了新的时间日期类 `ZonedDateTime`、`ZoneOffset`、`LocalDateTime`、`ZonedDateTime` 和 `DateTimeFormatter`，处理时区问题更简单清晰。我们再用这些类配合一个完整的例子，来理解一下时间的解析和展示：

首先初始化上海、纽约和东京三个时区。我们可以使用 `ZoneId.of` 来初始化一个标准的时区，也可以使用 `ZoneOffset.ofHours` 通过一个 `offset`，来初始化一个具有指定时间差的自定义时区。

对于日期时间表示，`LocalDateTime` 不带有时区属性，所以命名为本地时区的日期时间；而 `ZonedDateTime`=`LocalDateTime`+`ZoneId`，具有时区属性。因此，`LocalDateTime` 只能认为是一个时间表示，`ZonedDateTime` 才是一个有效的时间。在这里我们把 2020-01-02 22:00:00 这个时间表示，使用东京时区来解析得到一个 `ZonedDateTime`。

使用 `DateTimeFormatter` 格式化时间的时候，可以直接通过 `withZone` 方法直接设置格式化使用的时区。最后，分别以上海、纽约和东京三个时区来格式化这个时间输出：


```
1 //一个时间表示
2 String stringDate = "2020-01-02 22:00:00";
3 //初始化三个时区
4 ZoneId timeZoneSH = ZoneId.of("Asia/Shanghai");
5 ZoneId timeZoneNY = ZoneId.of("America/New_York");
6 ZoneId timeZoneJST = ZoneOffset.ofHours(9);
7 //格式化器
8 DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd I
9 ZonedDateTime date = ZonedDateTime.of(LocalDate.parse(stringDate, dateTime
10 //使用DateTimeFormatter格式化时间，可以通过withZone方法直接设置格式化使用的时区
```

[复制代码](#)



```
11 DateTimeFormatter outputFormat = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
12 System.out.println(timeZoneSH.getId() + outputFormat.withZone(timeZoneSH).format(now));
13 System.out.println(timeZoneNY.getId() + outputFormat.withZone(timeZoneNY).format(now));
14 System.out.println(timeZoneJST.getId() + outputFormat.withZone(timeZoneJST).format(now));
```

可以看到，相同的时区，经过解析存进去和读出来的时间表示是一样的（比如最后一行）；而对于不同的时区，比如上海和纽约，最后输出的本地时间不同。+9 小时时区的晚上 10 点，对于上海是 +8 小时，所以上海本地时间是晚上 9 点；而对于纽约是 -5 小时，差 14 小时，所以是早上 8 点：

 复制代码

```
1 Asia/Shanghai2020-01-02 21:00:00 +0800
2 America/New_York2020-01-02 08:00:00 -0500
3 +09:002020-01-02 22:00:00 +0900
```

到这里，我来小结下。要正确处理国际化时间问题，我推荐使用 Java 8 的日期时间类，即使用 `ZonedDateTime` 保存时间，然后使用设置了 `ZoneId` 的 `DateTimeFormatter` 配合 `ZonedDateTime` 进行时间格式化得到本地时间表示。这样的划分十分清晰、细化，也不容易出错。

接下来，我们继续看看对于日期时间的格式化和解析，使用遗留的 `SimpleDateFormat`，会遇到哪些问题。

## 日期时间格式化和解析

每到年底，就有很多开发同学踩时间格式化的坑，比如“这明明是一个 2019 年的日期，**如何使用 `SimpleDateFormat` 格式化后就提前跨年了**”。我们来重现一个这个问题。

初始化一个 `Calendar`，设置日期时间为 2019 年 12 月 29 日，使用大写的 `YYYY` 来初始化 `SimpleDateFormat`：


 复制代码

```
1 Locale.setDefault(Locale.SIMPLIFIED_CHINESE);
2 System.out.println("defaultLocale:" + Locale.getDefault());
3 Calendar calendar = Calendar.getInstance();
4 calendar.set(2019, Calendar.DECEMBER, 29, 0, 0, 0);
5 SimpleDateFormat YYYY = new SimpleDateFormat("YYYY-MM-dd");
```

```
6 System.out.println("格式化: " + YYYY.format(calendar.getTime()));
7 System.out.println("weekYear:" + calendar.getWeekYear());
8 System.out.println("firstDayOfWeek:" + calendar.getFirstDayOfWeek());
9 System.out.println("minimalDaysInFirstWeek:" + calendar.getMinimalDaysInFirstWeek());
```

得到的输出却是 2020 年 12 月 29 日:

```
1 defaultLocale:zh_CN
2 格式化: 2020-12-29
3 weekYear:2020
4 firstDayOfWeek:1
5 minimalDaysInFirstWeek:1
```

 复制代码

出现这个问题的原因在于，这位同学混淆了 `SimpleDateFormat` 的各种格式化模式。JDK 的 [文档](#)中有说明：小写 `y` 是年，而大写 `Y` 是 week year，也就是所在的周属于哪一年。

一年第一周的判断方式是，从 `getFirstDayOfWeek()` 开始，完整的 7 天，并且包含那一年至少 `getMinimalDaysInFirstWeek()` 天。这个计算方式和区域相关，对于当前 `zh_CN` 区域来说，2020 年第一周的条件是，从周日开始的完整 7 天，2020 年包含 1 天即可。显然，2019 年 12 月 29 日周日到 2020 年 1 月 4 日周六是 2020 年第一周，得出的 week year 就是 2020 年。

如果把区域改为法国：

```
1 Locale.setDefault(Locale.FRANCE);
```

 复制代码

那么 week yeay 就还是 2019 年，因为一周的第一天从周一开始算，2020 年的第一周是 2019 年 12 月 30 日周一开始，29 日还是属于去年：

```
1 defaultLocale:fr_FR
2 格式化: 2019-12-29
3 weekYear:2019
4 firstDayOfWeek:2
5 minimalDaysInFirstWeek:4
```

 复制代码



这个案例告诉我们，没有特殊需求，针对年份的日期格式化，应该一律使用 “y” 而非 “Y” 。

除了格式化表达式容易踩坑外，SimpleDateFormat 还有两个著名的坑。

第一个坑是，**定义的 static 的 SimpleDateFormat 可能会出现线程安全问题**。比如像这样，使用一个 100 线程的线程池，循环 20 次把时间格式化任务提交到线程池处理，每个任务中又循环 10 次解析 2020-01-01 11:12:13 这样一个时间表示：

 复制代码

```
1  ExecutorService threadPool = Executors.newFixedThreadPool(100);
2  for (int i = 0; i < 20; i++) {
3      //提交20个并发解析时间的任务到线程池，模拟并发环境
4      threadPool.execute(() -> {
5          for (int j = 0; j < 10; j++) {
6              try {
7                  System.out.println(simpleDateFormat.parse("2020-01-01 11:12:13
8                  } catch (ParseException e) {
9                      e.printStackTrace();
10                 }
11             }
12         });
13     }
14     threadPool.shutdown();
15     threadPool.awaitTermination(1, TimeUnit.HOURS);
```

运行程序后大量报错，且没有报错的输出结果也不正常，比如 2020 年解析成了 1212 年：

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_211.jdk/Contents/Home/bin/java ...
Exception in thread "pool-1-thread-2" Exception in thread "pool-1-thread-4" java.lang.NumberFormatException: For input string: "101.E1012E"
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
    at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.lang.Double.parseDouble(Double.java:538)
    at java.text.DigitList.getDouble(DigitList.java:169)
    at java.text.DecimalFormat.parse(DecimalFormat.java:2089)Wed Jan 01 11:12:13 CST 2020
Wed Jan 01 11:12:13 CST 2020
Sun Jan 01 11:12:13 CST 1212
Wed Jan 01 11:12:13 CST 2020

    at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:1869)
    at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
    at java.text.DateFormat.parse(DateFormat.java:364)
    at org.geekbang.time.commonmistakes.datetime.dateformat.CommonMistakesApplication.lambda$wrong2$1(CommonMistakesApplication.java:90)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)
Wed Jan 01 11:12:13 CST 2020
Wed Jan 01 11:12:13 CST 2020
Wed Jan 01 11:12:13 CST 2020
Wed Jan 01 11:12:13 CST 2020
java.lang.NumberFormatException: For input string: "101.E1012E2"
Wed Jan 01 11:12:13 CST 2020
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
Wed Jan 01 11:12:13 CST 2020
    at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.lang.Double.parseDouble(Double.java:538)Sun Jan 01 11:12:13 CST 1111

    at java.text.DigitList.getDouble(DigitList.java:169)
    at java.text.DecimalFormat.parse(DecimalFormat.java:2089)
    at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:1869)
Wed Jan 01 11:12:13 CST 2020

```


SimpleDateFormat 的作用是定义解析和格式化日期时间的模式。这，看起来这是一次性的工作，应该复用，但它的解析和格式化操作是非线程安全的。我们来分析一下相关源码：

SimpleDateFormat 继承了 DateFormat，DateFormat 有一个字段 Calendar；

SimpleDateFormat 的 parse 方法调用 CalendarBuilder 的 establish 方法，来构建 Calendar；

establish 方法内部先清空 Calendar 再构建 Calendar，整个操作没有加锁。

显然，如果多线程池调用 parse 方法，也就意味着多线程在并发操作一个 Calendar，可能会产生一个线程还没来得及处理 Calendar 就被另一个线程清空了的情况：

 复制代码

```

1 public abstract class DateFormat extends Format {
2     protected Calendar calendar;
3 }
4 public class SimpleDateFormat extends DateFormat {
5     @Override
6     public Date parse(String text, ParsePosition pos)
7     {
8         CalendarBuilder calb = new CalendarBuilder();
9         parsedDate = calb.establish(calendar).getTime();
10        return parsedDate;
11    }
12 }
13
14 class CalendarBuilder {

```

```

15     Calendar establish(Calendar cal) {
16         ...
17         cal.clear();//清空
18
19         for (int stamp = MINIMUM_USER_STAMP; stamp < nextStamp; stamp++) {
20             for (int index = 0; index <= maxFieldIndex; index++) {
21                 if (field[index] == stamp) {
22                     cal.set(index, field[MAX_FIELD + index]);//构建
23                     break;
24                 }
25             }
26         }
27         return cal;
28     }
29 }

```

format 方法也类似，你可以自己分析。因此只能在同一个线程复用 SimpleDateFormat，比较好的解决方式是，通过 ThreadLocal 来存放 SimpleDateFormat：

 复制代码

```

1 private static ThreadLocal<SimpleDateFormat> threadSafeSimpleDateFormat = Thre;

```

第二个坑是，**当需要解析的字符串和格式不匹配的时候，SimpleDateFormat 表现得很宽容**，还是能得到结果。比如，我们期望使用 yyyyMM 来解析 20160901 字符串：

 复制代码

```

1 String dateString = "20160901";
2 SimpleDateFormat dateFormat = new SimpleDateFormat("yyyyMM");
3 System.out.println("result:" + dateFormat.parse(dateString));

```

居然输出了 2091 年 1 月 1 日，原因是把 0901 当成了月份，相当于 75 年：

 复制代码


```

1 result:Mon Jan 01 00:00:00 CST 2091

```


对于 SimpleDateFormat 的这三个坑，我们使用 Java 8 中的 DateTimeFormatter 就可以避过去。首先，使用 DateTimeFormatterBuilder 来定义格式化字符串，不用去记忆使用

大写的 Y 还是小写的 Y，大写的 M 还是小写的 m：

 复制代码

```
1 private static DateTimeFormatter dateTimeFormatter = new DateTimeFormatterBuilder
2     .appendValue(ChronoField.YEAR) //年
3     .appendLiteral("/")
4     .appendValue(ChronoField.MONTH_OF_YEAR) //月
5     .appendLiteral("/")
6     .appendValue(ChronoField.DAY_OF_MONTH) //日
7     .appendLiteral(" ")
8     .appendValue(ChronoField.HOUR_OF_DAY) //时
9     .appendLiteral(":")
10    .appendValue(ChronoField.MINUTE_OF_HOUR) //分
11    .appendLiteral(":")
12    .appendValue(ChronoField.SECOND_OF_MINUTE) //秒
13    .appendLiteral(".")
14    .appendValue(ChronoField.MILLI_OF_SECOND) //毫秒
15    .toFormatter();
```

其次，DateTimeFormatter 是线程安全的，可以定义为 static 使用；最后，DateTimeFormatter 的解析比较严格，需要解析的字符串和格式不匹配时，会直接报错，而不会把 0901 解析为月份。我们测试一下：

 复制代码

```
1 //使用刚才定义的DateTimeFormatterBuilder构建的DateTimeFormatter来解析这个时间
2 LocalDateTime localDateTime = LocalDateTime.parse("2020/1/2 12:34:56.789", dateTimeFormatter);
3 //解析成功
4 System.out.println(localDateTime.format(dateTimeFormatter));
5 //使用yyyyMM格式解析20160901是否可以成功呢？
6 String dt = "20160901";
7 DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyyMM");
8 System.out.println("result:" + dateTimeFormatter.parse(dt));
```

输出日志如下：


 复制代码

```
1 2020/1/2 12:34:56.789
2 Exception in thread "main" java.time.format.DateTimeParseException: Text '20160901'
3   at java.time.format.DateTimeFormatter.parseResolved0(DateTimeFormatter.java:1154)
4   at java.time.format.DateTimeFormatter.parse(DateTimeFormatter.java:1177)
5   at org.geekbang.time.commonmistakes.datetime.dateformat.CommonMistakesApplication.main(CommonMistakesApplication.java:10)
6   at org.geekbang.time.commonmistakes.datetime.dateformat.CommonMistakesApplication.main(CommonMistakesApplication.java:10)
```

到这里我们可以发现，使用 Java 8 中的 `DateTimeFormatter` 进行日期时间的格式化和解析，显然更让人放心。那么，对于日期时间的运算，使用 Java 8 中的日期时间类会不会更简单呢？

## 日期时间的计算

关于日期时间的计算，我先和你说一个常踩的坑。有些同学喜欢直接使用时间戳进行时间计算，比如希望得到当前时间之后 30 天的时间，会这么写代码：直接把 `new Date().getTime` 方法得到的时间戳加 30 天对应的毫秒数，也就是 30 天 \* 1000 毫秒 \* 3600 秒 \* 24 小时：

 复制代码

```
1 Date today = new Date();
2 Date nextMonth = new Date(today.getTime() + 30 * 1000 * 60 * 60 * 24);
3 System.out.println(today);
4 System.out.println(nextMonth);
```

得到的日期居然比当前日期还要早，根本不是晚 30 天的时间：

 复制代码

```
1 Sat Feb 01 14:17:41 CST 2020
2 Sun Jan 12 21:14:54 CST 2020
```

出现这个问题，**其实是因为 int 发生了溢出**。修复方式就是把 30 改为 30L，让其成为一个 long：

 复制代码

```
1 Date today = new Date();
2 Date nextMonth = new Date(today.getTime() + 30L * 1000 * 60 * 60 * 24);
3 System.out.println(today);
4 System.out.println(nextMonth);
```

这样就可以得到正确结果了：

```
1 Sat Feb 01 14:17:41 CST 2020
2 Mon Mar 02 14:17:41 CST 2020
```

[复制代码](#)

不难发现，手动在时间戳上进行计算操作的方式非常容易出错。对于 Java 8 之前的代码，我更建议使用 Calendar：

```
1 Calendar c = Calendar.getInstance();
2 c.setTime(new Date());
3 c.add(Calendar.DAY_OF_MONTH, 30);
4 System.out.println(c.getTime());
```

[复制代码](#)

使用 Java 8 的日期时间类型，可以直接进行各种计算，更加简洁和方便：

```
1 LocalDateTime localDateTime = LocalDateTime.now();
2 System.out.println(localDateTime.plusDays(30));
```

[复制代码](#)

并且，**对日期时间做计算操作，Java 8 日期时间 API 会比 Calendar 功能强大很多。**

第一，可以使用各种 minus 和 plus 方法直接对日期进行加减操作，比如如下代码实现了减一天和加一天，以及减一个月和加一个月：

```
1 System.out.println("//测试操作日期");
2 System.out.println(LocalDate.now()
3     .minus(Period.ofDays(1))
4     .plus(1, ChronoUnit.DAYS)
5     .minusMonths(1)
6     .plus(Period.ofMonths(1)));
```

[复制代码](#)

可以得到：

```
1 //测试操作日期
2 2020-02-01
```

[复制代码](#)



第二，还可以通过 with 方法进行快捷时间调节，比如：

使用 TemporalAdjusters.firstDayOfMonth 得到当前月的第一天；

使用 TemporalAdjusters.firstDayOfYear() 得到当前年的第一天；

使用 TemporalAdjusters.previous(DayOfWeek.SATURDAY) 得到上一个周六；

使用 TemporalAdjusters.lastInMonth(DayOfWeek.FRIDAY) 得到本月最后一个周五。

 复制代码

```
1 System.out.println("//本月的第一天");
2 System.out.println(LocalDate.now().with(TemporalAdjusters.firstDayOfMonth()));
3
4 System.out.println("//今年的程序员日");
5 System.out.println(LocalDate.now().with(TemporalAdjusters.firstDayOfYear()).plusDays(100));
6
7 System.out.println("//今天之前的一个周六");
8 System.out.println(LocalDate.now().with(TemporalAdjusters.previous(DayOfWeek.SATURDAY)));
9
10 System.out.println("//本月最后一个工作日");
11 System.out.println(LocalDate.now().with(TemporalAdjusters.lastInMonth(DayOfWeek.FRIDAY)));
```

输出如下：

 复制代码

```
1 //本月的第一天
2 2020-02-01
3 //今年的程序员日
4 2020-09-12
5 //今天之前的一个周六
6 2020-01-25
7 //本月最后一个工作日
8 2020-02-28
```


第三，可以直接使用 lambda 表达式进行自定义的时间调整。比如，为当前时间增加 100 天以内的随机天数：

 复制代码

```
1 System.out.println(LocalDate.now().with(temporal -> temporal.plus(ThreadLocalRandom.current().nextInt(100), DayOfWeek.FRIDAY)));
```

得到：

```
1 2020-03-15
```

 复制代码


除了计算外，还可以判断日期是否符合某个条件。比如，自定义函数，判断指定日期是否是家庭成员的生日：

```
1 public static Boolean isFamilyBirthday(TemporalAccessor date) {
2     int month = date.get(MONTH_OF_YEAR);
3     int day = date.get(DAY_OF_MONTH);
4     if (month == Month.FEBRUARY.getValue() && day == 17)
5         return Boolean.TRUE;
6     if (month == Month.SEPTEMBER.getValue() && day == 21)
7         return Boolean.TRUE;
8     if (month == Month.MAY.getValue() && day == 22)
9         return Boolean.TRUE;
10    return Boolean.FALSE;
11 }
```

 复制代码

然后，使用 query 方法查询是否匹配条件：

```
1 System.out.println("//查询是否是今天要举办生日");
2 System.out.println(LocalDate.now().query(CommonMistakesApplication::isFamilyBi
```

 复制代码

使用 Java 8 操作和计算日期时间虽然方便，但计算两个日期差时可能会踩坑：**Java 8 中有一个专门的类 Period 定义了日期间隔，通过 Period.between 得到了两个 LocalDate 的差，返回的是两个日期差几年零几月零几天。如果希望得知两个日期之间差几天，直接调用 Period 的 getDays() 方法得到的只是最后的“零几天”，而不是算总的间隔天数。**

比如，计算 2019 年 12 月 12 日和 2019 年 10 月 1 日的日期间隔，很明显日期差是 2 个月零 11 天，但获取 getDays 方法得到的结果只是 11 天，而不是 72 天：

[复制代码](#)

```
1 System.out.println("//计算日期差");
2 LocalDate today = LocalDate.of(2019, 12, 12);
3 LocalDate specifyDate = LocalDate.of(2019, 10, 1);
4 System.out.println(Period.between(specifyDate, today).getDays());
5 System.out.println(Period.between(specifyDate, today));
6 System.out.println(ChronoUnit.DAYS.between(specifyDate, today));
```

可以使用 `ChronoUnit.DAYS.between` 解决这个问题：

[复制代码](#)

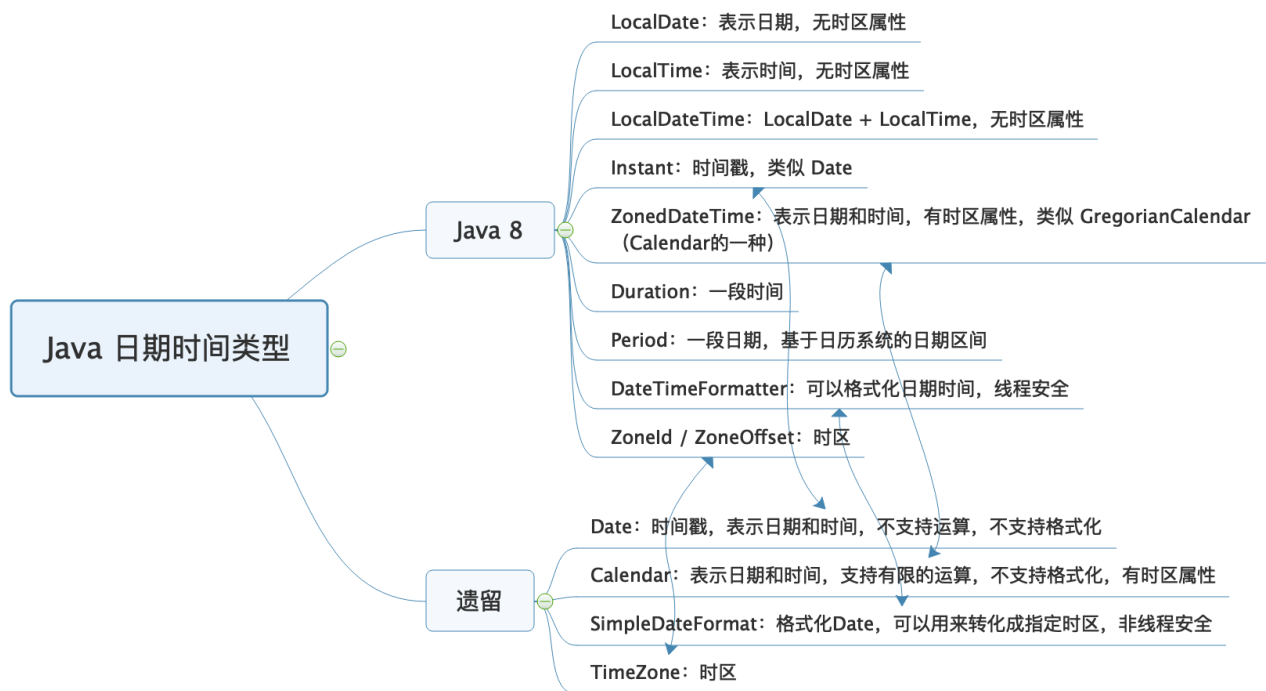
```
1 //计算日期差
2 11
3 P2M11D
4 72
```

从日期时间的时区到格式化再到计算，你是不是体会到 Java 8 日期时间类的强大了呢？

## 重点回顾

今天，我和你一起看了日期时间的初始化、时区、格式化、解析和计算的问题。我们看到，使用 Java 8 中的日期时间包 `Java.time` 的类进行各种操作，会比使用遗留的 `Date`、`Calender` 和 `SimpleDateFormat` 更简单、清晰，功能也更丰富、坑也比较少。

如果有条件的话，我还是建议全面改为使用 Java 8 的日期时间类型。我把 Java 8 前后的日期时间类型，汇总到了一张思维导图上，图中箭头代表的是新老类型在概念上等价的类型：



这里有个误区是，认为 `java.util.Date` 类似于新 API 中的 `LocalDateTime`。其实不是，虽然它们都没有时区概念，但 `java.util.Date` 类是因为使用 UTC 表示，所以没有时区概念，其本质是时间戳；而 `LocalDateTime`，严格上可以认为是一个日期时间的表示，而不是一个时间点。

因此，在把 `Date` 转换为 `LocalDateTime` 的时候，需要通过 `Date` 的 `toInstant` 方法得到一个 UTC 时间戳进行转换，并需要提供当前的时区，这样才能把 UTC 时间转换为本地日期时间（的表示）。反过来，把 `LocalDateTime` 的时间表示转换为 `Date` 时，也需要提供时区，用于指定是哪个时区的时间表示，也就是先通过 `atZone` 方法把 `LocalDateTime` 转换为 `ZonedDateTime`，然后才能获得 UTC 时间戳：

复制代码

```
1 Date in = new Date();
2 LocalDateTime ldt = LocalDateTime.ofInstant(in.toInstant(), ZoneId.systemDefault());
3 Date out = Date.from(ldt.atZone(ZoneId.systemDefault()).toInstant());
```

很多同学说使用新 API 很麻烦，还需要考虑时区的概念，一点都不简洁。但我通过这篇文章要和你说的，并不是因为 API 需要设计得这么繁琐，而是 UTC 时间要变为当地时间，必须考虑时区。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

## 思考与讨论

1. 我今天多次强调 Date 是一个时间戳，是 UTC 时间、没有时区概念，为什么调用其 toString 方法会输出类似 CST 之类的时区字样呢？
2. 日期时间数据始终要保存到数据库中，MySQL 中有两种数据类型 datetime 和 timestamp 可以用来保存日期时间。你能说说它们的区别吗，它们是否包含时区信息呢？

对于日期和时间，你还遇到过什么坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你  
攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 序列化：一来一回你还是原来的你吗？

下一篇 17 | 别以为“自动挡”就不可能出现OOM

## 精选留言 (15)

 写留言



2020-04-16

思考题1:

根本原因在于toString的源代码:

```
sb.append(zi.getDisplayName(date.isDaylightTime(), TimeZone.SHORT, Locale.US));  
// zzz 这一行。
```

Date的toString实际上, 是新建一个StringBuilder, 然后根据Date对象里的年月日周, ...

展开 ▾



6



**Darren**

2020-04-16

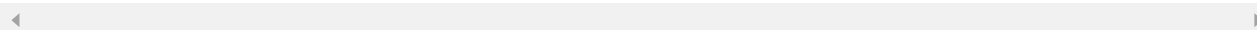
试着回到下问题:

第一个:

Date的toString()方法处理的, 同String中有BaseCalendar.Date date = normalize();  
而normalize中进行这样处理cdate = (BaseCalendar.Date) cal.getCalendarDate(fastTime, TimeZone.getDefaultRef()); ...

展开 ▾

作者回复: 很全面



3



2020-04-16

对于时间, 我个人的理解和目前的使用经验是——能用时间戳就用时间戳。

时间戳有几个优势:

- 1, 便于比较和排序, 无论数据库还是后台业务中都是如此。
- 2, 也比较便于计算, 虽然文中提到了Long的问题, 但是, 我认为L的问题的根本在于Long类型的理解, 不是时间戳这个业务的问题。对Long的基础比较好了之后, 也就足以应...

展开 ▾

作者回复: 保存和传输用时间戳的确比较方便, 用带有时区的字面量时间表示字符串, 还有格式不统一的问题



2



**pedro**

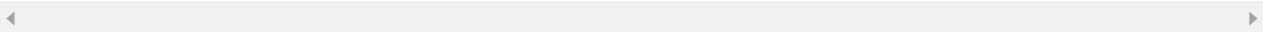
2020-04-16

第一个问题, 虽然 Date 本质是一个时间戳没有时区的概念, 但是在 toString 的时候为了可读性会推测当前时区, 如果得不到就会使用 GMT。



展开 ▾

作者回复: 是



2



2020-04-16

思考题2:

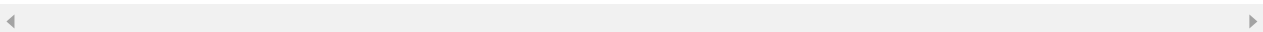
说实话，数据库相关知识是我的弱项。

查了一下，大概是TIMESTAMP包含了时区信息，而DATETIME不包含。另外有一个是，我印象中5.7之后的mysql版本，最多只能有一个TIMESTAMP的字段。这也算是个区别吧。

展开 ▾

作者回复: <https://dev.mysql.com/doc/refman/5.6/en/timestamp-initialization.html>

你说的应该是『As of MySQL 5.6.5, TIMESTAMP and DATETIME columns can be automatically initialized and updated to the current date and time (that is, the current timestamp). Before 5.6.5, this is true only for TIMESTAMP, and for at most one TIMESTAMP column per table. The following notes first describe automatic initialization and updating for MySQL 5.6.5 and up, then the differences for versions preceding 5.6.5.』这个问题，其实并不是指只能有一个TIMESTAMP 列，而是只能有一个TIMESTAMP列使用CURRENT\_TIMESTAMP来初始化或自动更新时间戳



1



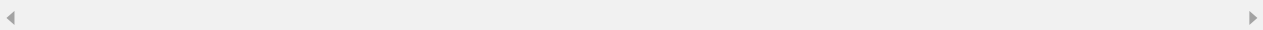
pedro

2020-04-16

第二个问题，timestamp 会把传入的时间转化为 UTC 即时间戳进行存储，而 datetime 也直接将传入的时间存储。

展开 ▾

作者回复: 大方向对，可以再考虑一下我们如何选择



1



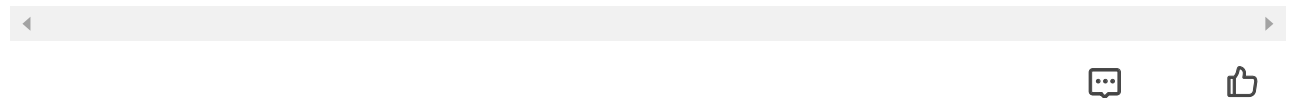
俊柱

2020-04-22

老师，我有一个问题困扰已久，希望能够解答一下，目前我们对外输出的 API，时间都是时间戳的形式，内部系统的交互，时间也是时间戳的形式。那我用 Instant 去映射数据库的 Timestamp/DateTime 字段，会不会更好？否则的话，需要在多处都要注意 LocalDateTime 和时间戳的相互转换（比如 redis 的序列化反序列化，json 的序列化、反序列化）

展开 ∨

作者回复: 当然可以使用Instant映射，参考<https://i.stack.imgur.com/idLPT.png>



**Michael**

2020-04-19

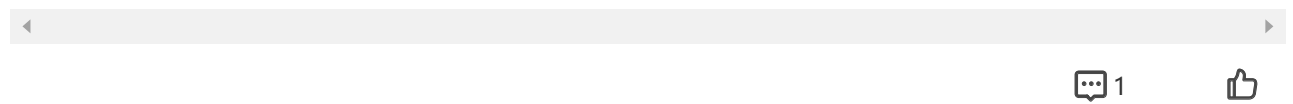
```
private static final DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
```

\*\*

\* 获取本周一开始时间...

展开 ∨

作者回复: 不太明白啥意思，代码中没有贴出获取本周日截止时间的办法，获取到下周一-1s即可



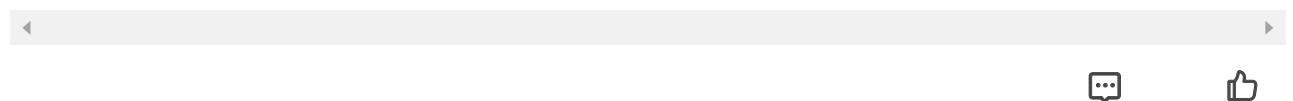
**Wiggle Wiggle**

2020-04-18

我在外企，公司大部分数据用的洛杉矶时间（带了夏令时），处理数据的时候要面对各种不带时区的string或datetime，无比酸爽

展开 ∨

作者回复: 处理数据的时候要面对各种不带时区的string或datetime。。。这显然是会有问题的

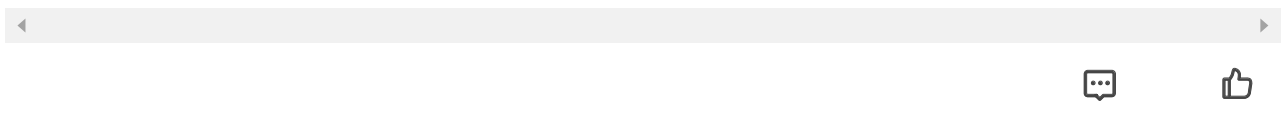


**wang**

2020-04-18

感觉LocalDateTime像是存了当前时区的zoneddatetime?

作者回复: 它就是一个时间表示, 没有保存时区, 理解为存了当前时区的ZonedDateTime不太准确



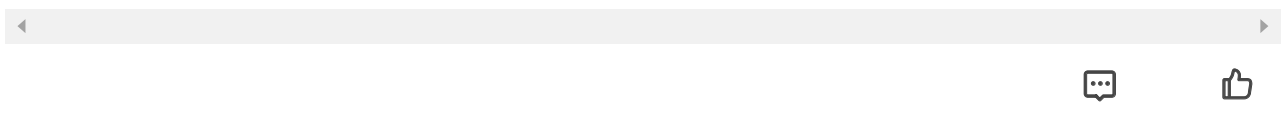
**刘大明**

2020-04-16

思考题1.是toString 方法里面if (zi != null) {  
    sb.append(zi.getDisplayName(date.isDaylightTime(), TimeZone.SHORT, Locale.US)); // zzz  
} else {  
    sb.append("GMT");...

展开 ▾

作者回复: 是



**jiarupc**

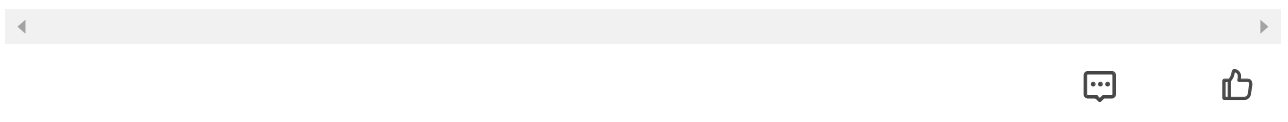
2020-04-16

在日期上, 我没踩过坑。

首先, 我做的都是国内项目, 不会出现时区的问题;  
其次, 对时间的应用真不多, 就是时间的保存、格式化、比较;  
...

展开 ▾

作者回复: 没关系没踩坑是好事情, 先了解一下



**Planeswalker23**

2020-04-16

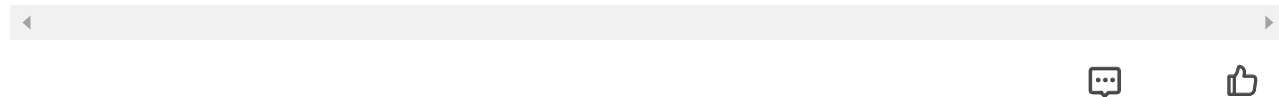
我虽然现在用的是jdk1.8, 但对于日期的操作一般还是习惯于用Date或long, 以后可以尝试用LocalDateTime等类。

思考题1

Date#toString方法中, 会将当前时间转化为BaseCalendar.Date类, 这个类有一个Zon...

展开 ▾

作者回复: 📖



eazonshaw

2020-04-16

问题二:

首先, 为了让docker容器的时间格式和宿主机一致, 可以在environment中添加TZ: Asia/Shanghai。

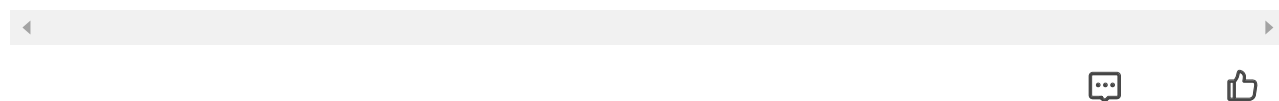
实验发现, 切换mysql的TIME\_ZONE到 “america/new\_york” 后, 发现datetime格式字段不发生变化, 而timestamp格式会换算成纽约时区时间, 所以timestamp格式的日期...

展开 ▾

作者回复: TIMESTAMP保存的时候根据当前时区转换为UTC, 查询的时候再根据当前时区从UTC转回来, 而DATETIME就是一个死的字符串时间 (仅仅对MySQL本身而言) 表示。有关mysql时间类型可以详细看一下这个ppt

<http://cdn.oreillystatic.com/en/assets/1/event/36/Time%20Zones%20and%20MySQL%20Presentation.pdf>

如果你的项目有国际化需求, 推荐使用时间戳, 并且需要确保你的应用服务器和数据库服务器设置了正确的匹配当地时区的时区配置 (其实, 即便你的项目没有国际化需求, 设置正确的需求, 至少是应用服务器和数据库服务器设置一致的时区, 也是需要的)



俊柱

2020-04-16

老师, 映射表的bean, 若数据库字段为 Timestamp, 那 java 的字段应该设为 ZonedDateTime 最为合理吗? 因为我看网上很多人都是用 LocalDateTime 进行映射

作者回复: 大多数时候项目没有全球化需求映射到本地时区即可, 可以使用LocalDateTime。

不过我们说datetime不包含时区, 是固定的时间表示仅仅是指MySQL本身。使用timestamp, 需要考虑Java进程的时区和MySQL连接的时区。而使用datetime类型, 则只需要考虑Java进程的时区 (因为MySQL datetime没有时区信息了, JDBC时间戳转换成MySQL datetime, 会根据MySQL的serverTimezone做一次转换)

具体你可能需要自己多做一些实验来理解，几个层面

- 1、mysql本身对于datetime和timestamp的区别
- 2、java应用和mysql交互时的关系

