为啥在编程的世界里，日期时间处理这么难?

做过开发的同学都有体会，日期时间的处理很麻烦，稍不注意就会出现日期格式不一样，或者时间差8小时。

那为何日期时间这么难处理呢？今天我们就来梳理一下在编程的世界里，为啥日期时间这么难处理。

我们先来熟悉几个概念
1、**时区（Timezone）**

由于各地的日出日落时间不同，所以把全球所有地区共分为24个时区，每个时区跨越360/24=15个经度，比如伦敦位于北京的西面，那么当北京的太阳已经升起的时候，伦敦还要再过 8 小时才能迎来黎明。也就是说，伦敦比北京晚 8 小时。而东京位于北京的东面，所以东京的日出比北京早 1 小时。

一定有人听过中国的时区是东八区，那这个东八区到底是啥意思呢?

以本初子午线（即0度经线，也称格林尼治子午线）为基准，向东每跨越15度经度就增加一个时区，向西则相应减去一个时区

中国位于东经约73度至135度之间，理论上跨越了五个时区，即东五区至东九区。然而，为了便于全国范围内的行政管理、交通调度、通信交流以及社会生活的统一协调，中国选择了东经120度所在的东八区作为全国统一的标准时间，即“北京时间”

如果我们想知道当北京是中午 12:00 的时候，东京是什么时间，可以先用 12:00 减去当前时区 +08:00，换算成伦敦时间 04:00，再加上目标时区 +09:00，就得到了东京时间 13:00

2、**格林威治时间（Greenwich Mean Time, GMT）**

GMT是基于英国伦敦格林尼治天文台所在经线（本初子午线）的地方时。作为曾经的世界标准时间，它不考虑夏令时（DST）调整。现在通常用作UTC（协调世界时）的同义词，尽管在实际应用中可能仍有细微差别

3、**协调世界时（Coordinated Universal Time, UTC）**

UTC又称世界统一时间、世界标准时间、国际协调时间。是当前国际通用的时间标准，它基于原子钟的测量结果，与GMT基本相同，但在需要插入闰秒以保持与地球自转的长期同步时，两者会有所区别。UTC时间通常以“UTC”后跟时分秒表示，如“2024-04-26T12:30:00Z”

4、**夏令时（Daylight Saving Time, DST）**

一到夏天，白天就变得很长，特别是高纬度地区会更明显，到了北极或南极，太阳整天都不会落下去，这就是极昼。为了充分利用大自然的馈赠，有些地方会实行夏令时，也就是说到了夏天，就人为把表拨快一个小时，让人们早点起床、早点睡觉，这样可以节省一些照明的电费。中国曾经短暂实行过几年夏令时，不过后来认为它带来的负面影响超过收益，就取消了。但是世界上仍然有很多地方实施夏令时，当设计全球化应用的时候，必须得考虑它

5、**CST**

CST这个缩写它可以同时代表四个不同的时间：

* China Standard Time —— 中国标准时间（UTC+8）
* Central Standard Time (USA) —— 美国中央时区（UTC-6）
* Central Standard Time (Australia) —— 澳大利亚中央时区 （UTC+9）
* Cuba Standard Time UTC-4:00 —— 古巴标准时区（UTC-4）

因此，使用CST时需注意其具体指代哪个时区，以免产生混淆

6、**ISO 8601**

这是一个国际标准化组织（International Organization for Standardization, ISO）制定的日期和时间表示法。ISO 8601提供了一种清晰、无歧义的日期和时间格式，如“YYYY-MM-DD”表示日期，“HH:MM:SS”表示时间，两者结合为“YYYY-MM-DDTHH:MM:SS”（如“2024-04-26T12:30:00”）。该标准还支持时区表示，如“2024-04-26T12:30:00+01:00”表示在UTC基础上加1小时的时区时间

7、**时间戳（Timestamp）**

时间戳是一种精确记录某一事件发生时刻的方法，通常是一个从特定起点（如Unix时间戳的起点是1970年1月1日0时0分0秒（UTC））开始计数的整数或浮点数，单位通常是秒或毫秒。例如，Unix时间戳“1678882200”表示距离该起点的秒数。时间戳常用于计算机系统中记录事件发生的精确时间点，便于比较和排序

8、**网络时间协议（Network Time Protocol，NTP）**

NTP是一种网络协议，用于在分布式系统中同步各个设备的时钟，以确保所有参与通信的节点拥有高度一致的时间。NTP通过在客户端与NTP服务器之间交换时间戳信息，通过复杂的算法计算时钟偏差和网络延迟，从而调整客户端系统的时钟以接近服务器提供的准确时间。NTP对于确保网络应用中的事务完整性、日志记录准确性、分布式系统协作等具有重要意义

9、**本地时间**

本地时间是指某个特定地点当前所采用的时间，包括了时区偏移和可能的夏令时调整。例如，“北京时间”对应东八区时间，表示为“UTC+08:00”。本地时间通常用于日常生活和商业活动，如“2024年4月26日 10:30 AM（北京）”

怎么样？是不是看了上面的梳理，感觉有些头大，这还没有算上中国的农历、还有比如康熙、乾隆多少年，都可以表示日期和时间。

那这么多时间格式，你在做项目的时候不可能全部会用到，一般如果不涉及到国际业务，可以使用本地时间来简单化处理。

可以只用存储时间戳，不存储时区信息，这样可以依赖系统自带的时区来简单处理日期时间

Java中是如何处理日期时间的

在Java 8以前，日期和时间的处理主要依赖于`java.util.Date`和`java.util.Calendar`这两个类

`java.util.Date` 类

`java.util.Date`类实际上包含了日期和时间，但它的toString()方法默认输出的是带有当前时区的时间，而getYear(), getMonth(), getDate()等方法已经过时，并且月份是从0开始计数的（1表示二月，依此类推）

```
import java.util.Date;

public class OldDateExample {
    public static void main(String[] args) {
        // 获取当前日期和时间
        Date currentDate = new Date();

        // 打印当前日期和时间
        System.out.println("Current date and time (in default format): " + currentDate);

        // 转换为自定义格式，需要用到SimpleDateFormat类
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String formattedDate = sdf.format(currentDate);
        System.out.println("Current date and time (formatted): " + formattedDate);

        // 设置一个新的日期（此方式已过时，但仍可用于演示）
        Calendar cal = Calendar.getInstance();
        cal.set(2000, Calendar.JANUARY, 1, 0, 0, 0); // 注意月份是Calendar.JANUARY而不是1
        Date specificDate = cal.getTime();
        System.out.println("Specific date (old style): " + sdf.format(specificDate));
    }
}
```

`java.util.Calendar` 类

`java.util.Calendar` 类是用来处理日期和时间字段以及完成日期时间字段运算的抽象类，它为特定的日历系统提供了方法，如获取和设置日期字段，计算与日期相关的值等。但它不是线程安全的，并且API使用繁琐

```
import java.util.Calendar;
import java.text.SimpleDateFormat;

public class OldCalendarExample {
    public static void main(String[] args) {
        // 创建一个Calendar实例
        Calendar calendar = Calendar.getInstance();

        // 设置日期为2000年1月1日
        calendar.set(Calendar.YEAR, 2000);
        calendar.set(Calendar.MONTH, Calendar.JANUARY); // 注意月份需要减1，因为它是从0开始计数的
        calendar.set(Calendar.DAY_OF_MONTH, 1);
        calendar.set(Calendar.HOUR_OF_DAY, 0);
        calendar.set(Calendar.MINUTE, 0);
        calendar.set(Calendar.SECOND, 0);
        calendar.set(Calendar.MILLISECOND, 0);

        // 获取Date对象
        Date specificDate = calendar.getTime();

        // 将Date对象格式化为字符串
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String formattedDate = sdf.format(specificDate);

        System.out.println("Specific date using Calendar: " + formattedDate);
    }
}

```

这些类在设计和使用上存在一些不足，比如构造函数隐含了当前时区，日期时间的表示不够直观，而且API使用起来相对复杂

在实际开发中，强烈建议使用Java 8及以上版本的日期时间API，因为它们更简洁、直观且避免了许多老API中存在的陷阱

Java8新版日期时间处理方法

Java 8及以后版本中推荐使用的`java.time`包下的类来进行日期和时间的操作

1. **基础日期时间类**：

* `LocalDate`：表示日期，不包含任何时间信息，只年、月、日。
* `LocalTime`：表示时间，不包含日期，只小时、分钟、秒和纳秒。
* `LocalDateTime`：结合了日期和时间，依然不包含时区信息，用于表示具体的日期和时间点。
2. **时区相关类**：

* `ZonedDateTime`：包含日期、时间及时区信息，用于表示具体某一地点的日期和时间。
* `ZoneId`：表示时区，替代了旧版的`TimeZone`类。
3. **瞬时时间类**：

* `Instant`：表示时间线上某一特定的瞬间，通常对应 Unix 时间戳，内部以 epoch 秒和纳秒表示。
4. **日期时间格式化与解析**：

* `DateTimeFormatter`：用于日期和时间的格式化和解析，可自定义日期时间字符串的样式

示例代码：

```
// 创建 LocalDate 和 LocalTime
LocalDate today = LocalDate.now();
LocalTime currentTime = LocalTime.now();

// 创建 LocalDateTime
LocalDateTime now = LocalDateTime.now();

// 创建 ZonedDateTime
ZonedDateTime zdt = ZonedDateTime.now(ZoneId.of("Asia/Shanghai"));

// 创建 Instant
Instant currentInstant = Instant.now();

// 格式化和解析日期时间
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
```

```
String formattedNow = now.format(formatter);
LocalDateTime parsedDateTime = LocalDateTime.parse("2024-04-27 14:30:00", formatter);

// 日期时间操作
LocalDate tomorrow = today.plusDays(1);
LocalTime nextHour = currentTime.plusHours(1);

// 与时区相关的转换
ZonedDateTime utcTime = zdt.withZoneSameInstant(ZoneId.of("Etc/UTC"));
```

MySQL中日期时间的处理

我们实际开发当中，大部分时候都需要将日期时间写入MySQL数据库中进行持久化，那Java与MySQL之间是如何进行日期时间处理的呢?

在MySQL中处理日期时间，主要通过几种预定义的数据类型来存储和操作日期和时间值

* **DATE**：仅存储日期，格式为 'YYYY-MM-DD'，范围从'1000-01-01'到'9999-12-31'
* **TIME**：仅存储时间，格式为 'HH:MM:SS'
* **DATETIME**：存储日期和时间，格式为 'YYYY-MM-DD HH:MM:SS'，范围从'1000-01-01 00:00:00.000000'到'9999-12-31 23:59:59.999999'
* **TIMESTAMP**：类似于DATETIME，但自动更新到当前时间戳，并且受时区影响。其存储范围与DATETIME相似，但具体行为会根据MySQL的配置有所不同，特别是与服务器时区相关的自动调整

在Java 8及以后版本中，与MySQL的这些日期时间类型相对应，可以使用`java.time`包下的类

* **DATE** 对应 **LocalDate**：只包含日期部分，无时间信息。
* **TIME** 对应 **LocalTime**：只包含时间部分，无日期信息。
* **DATETIME** 可以对应 **LocalDateTime**：同时包含日期和时间，但不包含时区信息。如果需要时区信息，可以使用 **ZonedDateTime** 结合具体的时区信息。
* **TIMESTAMP** 在Java中通常与 **LocalDateTime** 或 **Instant** 相对应，具体取决于应用场景。如果TIMESTAMP用于记录带有时区的绝对时间

点，使用 **Instant** 更合适，因为它也是基于UTC时间的；如果是无时区的日期时间，则使用 **LocalDateTime**

注意：一般情况下，我们不需要存储时区，因为大部分项目都不涉及到国际化，只在国内开展业务的话，是不需要考虑时区的，依赖本地服务器的时区就可以减化日期时间的处理，但如果你的业务涉及到国际化，就必须考虑时区的问题了。

一般推荐将日期时间转换为UTC时区后存储到数据库，从数据库读取日期时间数据时，根据需要将其转换为应用程序的本地时区或用户指定的时区，以减少时区转换带来的问题。

你的项目中是如何处理日期时间和时区的问题呢？欢迎跟我留言来进行讨论！