,

ImportNew

- 导航条 -

JAVA 8：健壮、易用的时间/日期API

原文出处： About Rolandz

对很多应用来说，时间和日期的概念都是必须的。像生日，租赁期，事件的时间戳和商店营业时长，等等，都是基于时间和日期的；然而，**Java**却没有好的API来处理它们。在Java SE 8中，添加了一个新包：**java.time**，它提供了结构良好的API来处理时间和日期。

历史

在Java刚刚发布，也就是版本1.0的时候，对时间和日期仅有的支持就是**java.util.Date**类。大多数开发者对它的第一印象就是，它根本不代表一个“日期”。实际上，它只是简单的表示一个，从**1970-01-01Z**开始计时的，精确到毫秒的瞬时点。由于标准的**toString()**方法，按照JVM的默认时区输出时间和日期，有些开发人员把它误认为是时区敏感的。

在升级Java到1.1期间，**Date**类被认为是无法修复的。由于这个原因，**java.util.Calendar**类被添加了进来。悲剧的是，**Calendar**类并不比**java.util.Date**好多少。它们面临的部分问题是：

- 可变性。像时间和日期这样的类应该是不可变的。
- 偏移性。**Date**中的年份是从1900开始的，而月份都是从0开始的。
- 命名。**Date**不是“日期”，而**Calendar**也不真实“日历”。
- 格式化。格式化只对**Date**有用，**Calendar**则不行。另外，它也不是线程安全的。

大约在2001年，Joda-Time项目开始了。它的目的很简单，就是给Java提供一个高质量的时间和日期类库。尽管被耽搁了一段时间，它的1.0版还是被发布。很快，它就成为了广泛使用和流行的类库。随着时间的推移，有越来越多的需求，要在JDK中拥有一个像Joda-Time的这样类库。在来自巴西的Michael Nascimento Santos的帮助下，官方为JDK开发新的时间/日期API的进程：JSR-310，启动了。

综述

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

大多数开发者只会用到基础和**format**包，也可能会用到**temporal**包。因此，尽管有68个新的公开类型，大多数开发者，大概，将只会用到其中的三分之一。

日期

在新的API中，**LocalDate**是其中最重要的类之一。它是表示日期的不可变类型，不包含时间和时区。

“本地”，这个术语，我们对它的熟悉来自于Joda-Time。它原本出自ISO-8061的时间和日期标准，它和时区无关。实际上，本地日期只是日期的描述，例如“2014年4月5日”。特定的本地时间，因你在地球上的不同位置，开始于不同的时间

线。所以，澳大利亚的本地时间开始的比伦敦早10小时，比旧金山早18小时。

LocalDate被设计成，它的所有方法，都是常用方法：

```
LocalDate date = LocalDate.of(2014, Month.JUNE, 10);
int year = date.getYear(); // 2014
Month month = date.getMonth(); // 6月
int dom = date.getDayOfMonth(); // 10
DayOfWeek dow = date.getDayOfWeek(); // 星期二
int len = date.lengthOfMonth(); // 30 （6月份的天数）
boolean leap = date.isLeapYear(); // false （不是闰年）
```

在上面的例子中，我们看到日期使用工厂方法（所有的构造方法都是私有的）创建。然后用它查询了部分基本信息。注意：枚举类型，**Month**和**DayOfWeek**，被设计用来增强代码的可读性和可靠性。

在下面的例子中，我们来看看如何操作**LocalDate**的实例。由于它是不可变类型，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 date = LocalDate.of(2014, Month.JUNE, 10);
date = date.withYear(2015); // 2015-06-10
date = date.plusMonths(2); // 2015-08-10
date = date.minusDays(1); // 2015-08-09
```

上面这些都很简单，但有时我们需要对日期进行更复杂的修改。**java.time**包含了对此的处理机制：**TemporalAdjuster**类。**时间修改器**背后的设计思想是，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。API提供了一些通用的功能，你可以新增你自己的。修改器的使用很简单，但使用静态导入的方式，会让你更方便：

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

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

在**java.time**包中，像所有主要的时间/日期类一样，**LocalDate**类是固定于单个历法系统的：ISO-8601标准定义的历法。看见**修改器**的瞬间反应就是，这代码跟业务逻辑长的差不多啊！这对时间/日期类业务逻辑是很重的。我们最后想看的是多次手动修改日期。如果你的代码库里，有一个将会使用很多次的，对日期的通用操作，考虑把它做成修改器，然后告诉你的小组成员，把它当做一个写好的，测试过的组件，直接拿来用吧。

时间/日期值对象

值得花点时间弄清楚，是什么导致了**LocalDate**类成为值类型。值类型是这样一种简单的数据类型：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。**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。

大部分时间/日期类都应该是值类型的，**java.time**开发包印证了这一点。因此，我们没有理由去用==来判断2个**LocalDate**是不是相等，实际上，Javadoc也反对这样做。

对于值类型，想要更多了解的同学，可以参考我最近的文章，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。

不同的历法系统

在**java.time**包中，像所有主要的时间/日期类一样，**LocalDate**是固定于单个历法系统的：由ISO-8601标准定义。

ISO-8601历法系统是事实上的世界民用历法系统，也就是公历。平年有365天，闰年是366天。闰年的定义是：非世纪年，能被4整除；世纪年能被400整除。为了计算的一致性，公元1年的前一年被当做公元0年，以此类推。

采用这套历法，第一个影响就是，ISO-8601的日期不必跟**GregorianCalendar**一致。在**GregorianCalendar**中，凯撒历和格里高利历之间有一个转换日，一般默认在1582年10月15日。那天之前，用凯撒历：每4年一个闰年，没有例外。那天之后，用格里高利历，也就是公历，拥有稍微复杂点的闰年计算方式。

既然凯撒历和格里高利历之间的转换是个历史事实，那为什么新的**java.time**开发包不参照它呢？原因就是，现在使用历史日期的大部分Java应用程序，都是不正确的，继续下去，是个错误。这是为什么呢？当年，罗马的梵蒂冈，把历法从凯撒历

改换成格里高利历的时候，世界上大部分其他地区并没有更换历法。比如大英帝国，包括早期的美国，直到大约200后的1752年9月14日才换历法，沙俄直到1918年2月14日，而瑞典的历法转换更是一团糟。因此，实际上，对1918之前的日期，解释是相当多的；仅相信拥有单一转换日的**GregorianCalendar**，是不靠谱的。所以**LocalDate**中没有这种转换，就是一个合理的选择了。应用程序需要额外的上下文信息，才能在凯撒历和格里高利历间，精确的解释特定的历史日期。

第二个影响是，我们需要额外的一组类来帮助处理其他历法系统。**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，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。

每个历法系统都有自己的日期类，有**ThaiBuddhistDate**，**MinguoDate**，**JapaneseDate**，**HijrahDate**。它们应在对本地化有严重需求的应用中使用，比如为日本政府开发的系统。它们4个都继承了另外一个接口，**ChronoLocalDate**，可以让代码在不知道历法系统的情况下，去操作它们。虽然如此，但还是希望少用这个接口。

理解为什么少用**ChronoLocalDate**，对正确的使用整个**java.time**开发包很关键。真实情况是，当我们检视当前的应用，尽量以历法无关的方式来操作日期的大部分代码，都有问题。例如，你不能假定一年有12个月，而开发者都是这样认为的，并且增加12个月，他们就认为是增加了一年。你不能认为所有的月份都有相同的天数，比如，科普特人的历法，包括12个30天的月份，还有一个月仅有5天，或者6天。你也不能认为，下一年的年份就比现在的年份大了1年，比如日本历，在天皇换代时，会重新纪年，此时，还在那一年的年中（你甚至不能认为在同一个月的两天，是属于同一年的）。

在一个大型的系统中，唯一的以历法无关的方式开发的方法是：形成严格的代码审查制度，对日期和时间相关的每行代码都要做双重检查，以防偏向ISO历法系统。因此，推荐的做法是，在系统中，全部使用**LocalDate**，包括存储，操作和解释业务规则。仅有的，使用**ChronoLocalDate**的时候是，本地化的输入/输出，典型的做法是使用用户配置中首选的历法；即使如此，大多数应用并不需要那样的本地化级别。

如需更全面的了解，查看ChronoLocalDate的Javadoc。

时间

日期之后，下一个考虑的概念就是本地时间，**LocalTime**。典型的例子就是便利店的营业时间，例如从07:00到23:00（早上7点到晚上11点）。可能，在这个时间段营业的便利店，遍布整个美利坚，但是这个时间是本地化的，跟时区无关。

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。

LocalTime的用法跟**LocalDate**相似：

```
LocalTime time = LocalTime.of(20, 30);
int hour = date.getHour(); // 20
int minute = date.getMinute(); // 30
time = time.withSecond(6); // 20:30:06
time = time.plusMinutes(3); // 20:33:06
```

修改器机制同样适用于**LocalTime**，只是对它的复杂操作比较少。

时间和日期组合

下一个要考察的是**LocalDateTime**类。这个值类型只是**LocalDate**和**LocalTime**的简单组合。它表示一个跟时区无关的日期和时间。

LocalDateTime可以直接创建，或者组合时间和日期：

```
LocalDateTime dt1 = LocalDateTime.of(2014, Month.JUNE, 10, 20, 30);
LocalDateTime dt2 = LocalDateTime.of(date, time);
LocalDateTime dt3 = date.atTime(20, 30);
LocalDateTime dt4 = date.atTime(time);
```

第三和第四行使用**atTime()**方法，平滑地构造一个**LocalDateTime**实例。大部分的时间和日期类都有“**at**“方法：以这样的方式，把当前对象和其他对象组合，生成更复杂的对象。

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

时间点

在处理时间和日期的时候，我们通常会想到年，月，日，时，分，秒。然而，这只是时间的一个模型，是面向人类的。第二种通用模型是面向机器的，或者说是连续的。在此模型中，时间线中的一个点表示为一个很大的数。这有利于计算机处理。在UNIX中，这个数从1970年开始，以秒为的单位；同样的，在Java中，也是从1970年开始，但以毫秒为单位。

java.time包通过值类型**Instant**提供机器视图。**Instant**表示时间线上的一点，而不需要任何上下文信息，例如，时区。概念上讲，它只是简单的表示自1970年1月1日0时0分0秒（UTC）开始的秒数。因为**java.time**包是基于纳秒计算的，所以**Instant**的精度可以达到纳秒级。

```
Instant start = Instant.now();
// perform some calculation
Instant end = Instant.now();
assert end.isAfter(start);
```

Instant典型的用法是，当你需要记录事件的发生时间，而不需要记录任何有关时区信息时，存储和比较时间戳。它额很多有趣的地方在于，你不能对它做什么，而不是你能做什么。例如，下面的几行代码会抛出异常：

```
instant.get(ChronoField.MONTH_OF_YEAR);
instant.plus(6, ChronoUnit.YEARS);
```

抛出这些异常是因为，**Instant**只包含秒数和纳秒数，不提供处理人类意义上的时间单位。如果确实有需要，你需要额外提供时区信息。

时区

时区的概念由大英帝国开始采用。铁路的发明和通讯工具的改进，突然意味着人们的活动范围，大到跟太阳时的改变有很大的关系。在此之前，每个城镇和村庄，都通过太阳和日晷规定自己的时间。

下面的英国布鲁斯托交易所的时钟照片，显示了时区导致的最初混乱的一个例子。红色的指针显示格林威治时间，而黑色指针显示布鲁斯托时间，它们相差10分钟。

在技术的推动下，标准的时区系统慢慢演进，最终替代了老旧的本地太阳法计时。然而，关键的事实是，时区也是政治的产物。它们被用来显示对一个地区的政治控制，例如，最近的克里米亚时改为莫斯科时。一旦和政治挂钩，相关的规则常常不合逻辑。

时区规则，由一个发布IANA时区数据库的国际组织搜集和汇总。这些数据，包含地球上每个地区的标识和时区的历史变化。标识的格式类似”欧洲/伦敦“，或者”美洲/纽约“。

在**java.time**之前，我们用**TimeZone**表示时区，而现在，用**ZoneId**。它们有2个主要的不同。第一，**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；第二，实际的规则集在**ZoneRules**里，不在**ZoneId**中，通过**getRules()**方法可以获得。

时区的常见情况是从UTC/格林威治开始的一个固定偏移。我们通常在说时差的时候，会遇到它，例如，我们说纽约比伦敦晚5个小时。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。

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。

ZoneDateTime负责处理面向人类的（台历和挂钟上看到的）时间和面向机器的时间（始终连续增长的秒数）之间的转换。因此，你可以通过本地时间或时间点来创建**ZoneDateTime**实例：

```
ZoneId zone = ZoneId.of("Europe/Paris");

LocalDate date = LocalDate.of(2014, Month.JUNE, 10);
ZonedDateTime zdt1 = date.atStartOfDay(zone);

Instant instant = Instant.now();
ZonedDateTime zdt2 = instant.atZone(zone);
```

最恼人的时区问题之一就是夏令时。在夏令时中，从格林威治的偏移每年要调整两次（也许更多）；典型的做法是，春天调快时间，秋天再调回来。夏令时开始时，我们都需要手动调整家里的挂钟时间。这些调整，在**java.time**包中，叫偏移过渡。春天时，跟本地时间相比，缺了一段时间；相反，秋天时，有的时间会出现两次。

ZonedDateTime在它的工厂方法和控制方法中处理了这些。例如，在夏令时切换的那天，增加一天会增加逻辑上的一天：可能多于24小时，也可能少于24小时。同样的，方法**atStartOfDay()**之所以这样命名，是因为你不能假定它的处理结果就一定是午夜零点，夏令时开始的那天，一天是从午夜1点开始的。

下面是关于夏令时的最后一个小提示。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();
```

处于时间重叠时段时，使用这两个方法之一，你可以得到调整之前或调整之后的时间。在其他情况下，这两个方法是无效

的。

时间长度

到目前为止，我们讨论的时间/日期类以多种不同的方式表示时间线上的一个点。**java.time**还为时间长度额外提供了两个值类型。

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”。

它们可以作为参数，传给主要的时间/日期类的增加或减少时间的方法：

```
Period sixMonths = Period.ofMonths(6);
LocalDate date = LocalDate.now();
LocalDate future = date.plus(sixMonths);
```

解析和格式化

java.time.format包是专门用来格式化输出时间/日期的。这个包围绕**DateTimeFormatter**类和它的辅助创建类**DateTimeFormatterBuilder**展开。

静态方法加上**DateTimeFormatter**中的常量，是最通用的创建格式化器的方式。包括：

- 常用ISO格式常量，如**ISO_LOCAL_DATE**
- 字母模式，如**ofPattern(“dd/MM/uuuu”)**
- 本地化样式，如**ofLocalizedDate(FormatStyle.MEDIUM)**

很典型的，一旦有了格式化器，你可以把它传递给主要的时间/日期类的相关方法：

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

这把你从格式化器自己的格式化和解析方法中隔离开来。

如果你想控制格式化的语言环境，调用格式化器的**withLocale(Locale)**方法。相似的方式可以允许你控制格式化的历法系统、时区、十进制数和解析度。

如果你需要更多的控制权，查看**DateTimeFormatterBuilder**类吧，它允许你一步一步的构造更复杂的格式化器。它还提供大小写不敏感的解析，松散的解析，字符填充和可选的格式。

总结

Java 8中的java.time是一个新的、复杂的时间/日期API。它把Joda-Time中的设计思想和实现推向了更高的层次，让开发人员把**java.util.Date**和**Calendar**抛在了身后。是时候重新享受时间/日期编程的乐趣了。

- Oracle官方教程
- 非官方项目主页
- TreeTen-Extra项目，对Java Se类库的补充

可能感兴趣的文章

- Android手机中的“秘码”
- 讲故事，学（Java）设计模式—桥接模式
- Java高速、多线程虚拟内存
- 回到基础：封装集合
- 告别ORM
- 关于Java集合的小抄
- Java：过去、未来的互联网编程之王
- Java集合框架综述
- 跟我学 Spring 3（2.1）：IoC 基础
- 给jdk写注释系列之jdk1.6容器(9)：Strategy设计模式之Comparable&Comparator接口

发表评论

Name*

姓名

邮箱*

请填写邮箱

网站 (请以 http://开头)

请填写网站地址

评论内容*

请填写评论内容

(*) 表示必填项

提交评论

来自微博的评论

乐山ing 还可以输入140

表情 同步到微博 评论

43条评论

挣钱买肉 2015-2-7 23:15

@mywiz

回复

Japhia_栄 2015-2-2 22:51

@我的印象笔记

回复

百事可乐江鸟涛 2015-2-2 07:32

@我的印象笔记

回复

仙剑奇侠惊鸿再现 2015-2-2 07:14

@我的印象笔记

回复

马遇伯_2015 2015-2-2 06:41

java 新技能 ~…//@程序员的那些事:《JAVA 8：健壮、易用的时间/日期API》

回复

言西枣 2015-2-2 01:03

@我的印象笔记

回复

序_leo 2015-2-2 00:24

@我的印象笔记

回复

Miracle_淑阳 2015-2-2 00:22

@我的印象笔记

回复

山水oak 2015-2-1 23:49

@我的印象笔记

回复

飛凡_GZ 2015-2-1 23:39

@我的印象笔记

回复

更多

« Java 8：不要再用循环了 构造模式实践 »

,