Joda-Time 简介

既然无法摆脱时间，为何不设法简化时间处理？

任何企业应用程序都需要处理时间问题。应用程序需要知道当前的时间点和下一个时间点，有时它们还必须计算这两个时间点之间的路径。使用 JDK 完成这项任务将非常痛苦和繁琐。现在来看看 Joda Time，一个面向 Java™ 平台的易于使用的开源时间/日期库。正如您在本文中了解的那样，Joda-Time 轻松化解了处理日期和时间的痛苦和繁琐。

J Steven Perry 是一名软件开发人员、架构师和全能 Java 专家，他从 1991 年起就从事专业的软件开发。他的专业兴趣包括 JVM 的内部工作原理和 UML 建模，以及介于两者之间的所有内容。Steve 编写了从技术文档到 Java 代码等各种内容，并且对教学和培训也充满了热情。Steve 是 Java Management Extensions（O'Reilly）的作者，Java Enterprise Best Practices（O'Reilly）的合著者，并撰写了有关软件开发主题和 O'Reilly ShortCut: Log4J 的杂志文章。

2009 年 12 月 14 日

在编写企业应用程序时，我常常需要处理日期。并且在我的最新项目中 — 保险行业 — 纠正日期计算尤其重要。使用 `java.util.Calendar` 让我有些不安。如果您也曾使用这个类处理过日期/时间值，那么您就知道它使用起来有多麻烦。因此当我接触到 Joda-Time — 面向 Java 应用程序的日期/时间库的替代选择 — 我决定研究一下。其结果是：我很庆幸我这么做了。

Joda-Time 令时间和日期值变得易于管理、操作和理解。事实上，易于使用是 Joda 的主要设计目标。其他目标包括可扩展性、完整的特性集以及对多种日历系统的支持。并且 Joda 与 JDK 是百分之百可互操作的，因此您无需替换所有 Java 代码，只需要替换执行日期/时间计算的那部分代码。

本文将介绍并展示如何使用它。我将介绍以

多用
API 的
术上
a-

主题：

- 日期/时间替代库简介

- Joda 的关键概念
- 创建 Joda-Time 对象
- 以 Joda 的方式操作时间 style
- 以 Joda 的方式格式化时间

您可以 下载 演示这些概念的样例应用程序的源代码。

Time 名称表示相同的意思是一种误称。但在撰写本文之际，Joda-Time API 目前似乎是唯一处于活跃开发状态下的 Joda API。考虑到 Joda 大型项目的当前状态，我想将 Joda-Time 简称为 Joda 应该没什么问题。

Joda 简介

为什么要使用 Joda？考虑创建一个用时间表示的某个随意的时刻 — 比如，2000 年 1 月 1 日 0 时 0 分。我如何创建一个用时间表示这个瞬间的 JDK 对象？使用 `java.util.Date`？事实上这是行不通的，因为自 JDK 1.1 之后的每个 Java 版本的 Javadoc 都声明应当使用 `java.util.Calendar`。`Date` 中不赞成使用的构造函数的数量严重限制了您创建此类对象的途径。

然而，`Date` 确实有一个构造函数，您可以用来创建用时间表示某个瞬间的对象（除 “现在” 以外）。该方法使用距离 1970 年 1 月 1 日子时格林威治标准时间（也称为 *epoch*）以来的毫秒数作为一个参数，对时区进行校正。考虑到 Y2K 对软件开发企业的重要性，您可能会认为我已经记住了这个值 — 但是我没有。`Date` 也不过如此。

那么 `Calendar` 又如何呢？我将使用下面的方式创建必需的实例：

```
Calendar calendar = Calendar.getInstance();
calendar.set(2000, Calendar.JANUARY, 1, 0, 0, 0);
```

使用 Joda，代码应该类似如下所示：

```
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
```

这一行简单代码没有太大的区别。但是现在我将使问题稍微复杂化。假设我希望在这个日期上加上 90 天并输出结果。使用 JDK，我需要使用清单 1 中的代码：

清单 1. 以 JDK 的方式向某一个瞬间加上 9

```
Calendar calendar = Calendar.getInstance();
calendar.set(2000, Calendar.JANUARY, 1, 0, 0, 0);
SimpleDateFormat sdf =
  new SimpleDateFormat("E MM/dd/yyyy HH:mm:ss.SSS");
calendar.add(Calendar.DAY_OF_MONTH, 90);
System.out.println(sdf.format(calendar.getTime()));
```

使用 Joda，代码如清单 2 所示：

清单 2. 以 Joda 的方式向某一个瞬间加上 90 天并输出结果

```
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
System.out.println(dateTime.plusDays(90).toString("E MM/dd/yyyy HH:mm:ss.SSS"
```

两者之间的差距拉大了（Joda 用了两行代码，JDK 则是 5 行代码）。

现在假设我希望输出这样一个日期：距离 Y2K 45 天之后的某天在下一个月的当前周的最后一天的日期。坦白地说，我甚至不想使用 `Calendar` 处理这个问题。使用 JDK 实在太痛苦了，即使是简单的日期计算，比如上面这个计算。正是多年前的这样一个时刻，我第一次领略到 Joda-Time 的强大。使用 Joda，用于计算的代码如清单 3 所示：

清单 3. 改用 Joda

```
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
System.out.println(dateTime.plusDays(45).plusMonths(1).dayOfWeek()
  .withMaximumValue().toString("E MM/dd/yyyy HH:mm:ss.SSS");
```

清单 3 的输出为：

```
Sun 03/19/2000 00:00:00.000
```

如果您正在寻找一种易于使用的方式替代 JDK 日期处理，那么您真的应该考虑 Joda。如果不是这样的话，那么继续痛苦地使用 `Calendar` 完成所有日期计算吧。当您做到这一点后，您完全可以做到使用几把剪刀修建草坪并使用一把旧牙刷清洗您的汽车。

Joda 和 JDK 互操作性

JDK `Calendar` 类缺乏可用性，这一点很快就能体会到，而 Joda 弥补了这一不足。Joda 的设计者还做出了一个决定，我认为这是它取得成功的构建：JDK 互操作性。Joda 的类能够生成（但是，正如您将看到的一样，有时会采用一种比较迂回的方式）`java.util.Date` 的实例（和 `Calendar`）。这使您能够保留现有的依赖 JDK 的代码，但是又能够使用 Joda 处理复杂的日期/时间计算。

例如，完成 清单 3 中的计算后。我只需要[illegible]改就可以返回到 JDK 中：

清单 4. 将 Joda 计算结果插入到 JDK 对象

```
Calendar calendar = Calendar.getInstance();
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0,
System.out.println(dateTime.plusDays(45).plusMonths(1)
```

```
  .withMaximumValue().toString("E MM/dd/yyyy HH:mm:ss.SSS");
calendar.setTime(dateTime.toDate());
```

就是这么简单。我完成了计算，但是可以继续在 JDK 对象中处理结果。这是 Joda 的一个非常棒的特性。

Joda 的关键日期/时间概念

Joda 使用以下概念，它们可以应用到任何日期/时间库：

- 不可变性（Immutability）
- 瞬间性（Instant）
- 局部性（Partial）
- 年表（Chronology）
- 时区（Time zone）

我将针对 Joda 依次讨论每一个概念。

不可变性

我在本文讨论的 Joda 类具有不可变性，因此它们的实例无法被修改。（不可变类的一个优点就是它们是线程安全的）。我将向您展示的用于处理日期计算的 API 方法全部返回一个对应 Joda 类的新实例，同时保持原始实例不变。当您通过一个 API 方法操作 Joda 类时，您必须捕捉该方法的返回值，因为您正在处理的实例不能被修改。您可能对这种模式很熟悉；比如，这正是 `java.lang.String` 的各种操作方法的工作方式。

瞬间性

`Instant` 表示时间上的某个精确的时刻，使用从 epoch 开始计算的毫秒表示。这一定义与 JDK 相同，这就是为什么任何 Joda `Instant` 子类都可以与 JDK `Date` 和 `Calendar` 类兼容的原因。

更通用一点的定义是：一个*瞬间* 就是指时间线上只出现一次且唯一的一个时间点，并且这种日期结构只能以一种[illegible]次。

局部性

一个局部时间，正如我将在本文中将其称为[illegible]它指的是时间的一部分片段。瞬间性指定了与 epo[illegible]个精

确时刻，与此相反，局部时间片段指的是在时间上可以来回 “移动” 的一个时刻，这样它便可以应用于多个实例。比如，*6 月 2 日* 可以应用于任意一年的 6 月份（使用 Gregorian 日历）的第二天的任意瞬间。同样，*11:06 p.m.* 可以应用于任意一年的任意一天，并且每天只能使用一次。即使它们没有指定一个时间上的精确时刻，局部时间片段仍然是有用的。

我喜欢将局部时间片段看作一个重复周期中的一点，这样的话，如果我正在考虑的日期构建可以以一种有意义的方式出现多次（即重复的），那么它就是一个局部时间。

年表

Joda 本质 — 以及其设计核心 — 的关键就是*年表*（它的含义由一个同名抽象类捕捉）。从根本上讲，年表是一种日历系统 — 一种计算时间的特殊方式 — 并且是一种在其中执行日历算法的框架。受 Joda 支持的年表的例子包括：

- ISO（默认）
- Coptic
- Julian
- Islamic

Joda-Time 1.6 支持 8 种年表，每一种都可以作为特定日历系统的计算引擎。

时区

时区是值一个相对于英国格林威治的地理位置，用于计算时间。要了解事件发生的精确时间，还必须知道发生此事件的位置。任何严格的时间计算都必须涉及时区（或相对于 GMT），除非在同一个时区内发生了相对时间计算（即时这样时区也很重要，如果事件对于位于另一个时区的各方存在利益关系的话）。

`DateTimeZone` 是 Joda 库用于封装位置概 时间计算都可以在不涉及时区的情况下完成，但是

DateTimeZone 如何影响 Joda 的操作。默认时间，即从运行代码的机器的系统时钟检索到的时间，在大部分情况下被使用。

创建 Joda-Time 对象

现在，我将展示在采用该库时会经常遇到的一些 Joda 类，并展示如何创建这些类的实例。

本节中介绍的所有实现都具有若干构造函数，允许您初始化封装的日期/时间。它们可以分为 4 个类别：

- 使用系统时间。
- 使用多个字段指定一个瞬间时刻（或局部时间片段），达到这个特定实现所能支持的最细粒度的精确度。
- 指定一个瞬间时刻（或局部时间片段），以毫秒为单位。
- 使用另一个对象（例如，java.util.Date，或者是另一个 Joda 对象）。

可变的 Joda 类

我并不是可变实用类的粉丝；我只是认为它们的用例并不适合广泛使用。但是如果您认为您的确需要使用可变 Joda 类的话，本节的内容应当会对您的项目有帮助。Readable 和 ReadWritable API 之间的唯一区别在于 ReadWritable 类能够改变封装的日期/时间值，因此我在这里将不再介绍这一点。

我将在第一个类中介绍这些构造函数： DateTime。当您使用其他 Joda 类的相应构造函数时，也可以使用这里介绍的内容。

ReadableInstant

Joda 通过 ReadableInstant 类实现了瞬间性这一概念。表示时间上的不可变瞬间的 Joda 类都属于这个类的子类。（将这个类命名为 ReadOnlyInstant 可能更好，我认为这才是设计者需要传达的意思）。换句话说，ReadableInstant 表示时间上的某一个 不可修改的瞬间）。其中的两个子类分别为 DateTime 和 DateMidnight：

重载方法

如果您创建了一个 DateTime 的实例，并且没有提供 Chronology 或 DateTimeZone，Joda 将使用 ISOChronology（默认）和 DateTimeZone（来自系统设置）。然而，Joda 所有 超载方 Zone 为 用程序

- **DateTime**：这是最常用的一个类。它以毫秒级的精度封装时间上的某个瞬间时刻。DateTime 始终与 DateTimeZone 相关，如果您不指定它的话，它将被默认设置为运行代码的机器所在的时区。
可以使用多种方式构建 DateTime 对象。这个构造函数使用系统时间：

的的样例代码展示了如何使用这些超载方法（参见 下载）。我在这里不会再详细介绍它们，因为这些方法使用起来非常简单。然而，我建议您试着使用一下这个样例应用程序，看看编写您的应用程序代码有多么简单，这样您就可以随意地在 Joda 的 Chronology 和 DateTimeZone 之间切换，同时不会影响到代码的其余部分。

```
DateTime dateTime = new DateTime();
```

一般来讲，我会尽量避免使用系统时钟来初始化应用程序的实际，而是倾向于外部化设置应用程序代码使用的系统时间。样例应用程序执行以下代码：

```
DateTime dateTime = SystemFactory.getClock().getDateTime();
```

这使得使用不同日期/时间测试我的代码变得更加简单：我不需要修改代码来在应用程序中运行不同的日期场景，因为时间是在 SystemClock 实现的内部设置的，而不是在应用程序的内部。（我可以修改系统时间，但是那实在太痛苦了！）
下面的代码使用一些字段值构建了一个 DateTime 对象：

```
DateTime dateTime = new DateTime(
  2000, //year
  1,    // month
  1,    // day
  0,    // hour (midnight is zero)
  0,    // minute
  0,    // second
  0     // milliseconds
);
```

正如您所见，Joda 可以使您精确地控制创 方式，该对象表示时间上的某个特定的瞬间。每 与此类似的构造函数，您在此构造函数中指定 Jo 字段。您可以用它快速了解特定类在哪一种粒度

下一个构造函数将指定从 epoch 到某个时刻所经过的毫秒数。它根据 JDK Date 对象的毫秒值创建一个 DateTime 对象，其时间精度用毫秒表示，因为 epoch 与 Joda 是相同的：

```
java.util.Date jdkDate = obtainDateSomehow();
long timeInMillis = jdkDate.getTime();
DateTime dateTime = new DateTime(timeInMillis);
```

并且这个例子与前例类似，唯一不同之处是我在这里将 Date 对象直接传递给构造函数：

```
java.util.Date jdkDate = obtainDateSomehow();
dateTime = new DateTime(jdkDate);
```

Joda 支持使用许多其他对象作为构造函数的参数，用于创建 DateTime，如清单 5 所示：

清单 5. 直接将不同对象传递给 DateTime 的构造函数

```
// Use a Calendar
java.util.Calendar calendar = obtainCalendarSomehow();
dateTime = new DateTime(calendar);
// Use another Joda DateTime
DateTime anotherDateTime = obtainDateTimeSomehow();
dateTime = new DateTime(anotherDateTime);
// Use a String (must be formatted properly)
String timeString = "2006-01-26T13:30:00-06:00";
dateTime = new DateTime(timeString);
timeString = "2006-01-26";
dateTime = new DateTime(timeString);
```

注意，如果您准备使用 String（必须经过解析），您必须对其进行精确地格式化。参考 Javadoc，获得有关 Joda 的 ISODateTimeFormat 类的更多信息（参见 参考资料）。

- **DateMidnight**：这个类封装某个时区（通常为默认时区）在特定年/月/日的午夜时分的时刻。它基本上类似于 DateTime，不同之处在于时间部分总是为与该对象关联的特定 DateTimeZone 时区的午夜时分。

您将在本文看到的其他类都遵循与 ReadableInstant 类相同的模式（Joda Javadoc 将显示这些内容），因此为了节省篇幅，我将不会在以下小节介绍这些内容。

ReadablePartial

应用程序所需处理的日期问题并不全部都与 刻有关，因此您可以处理一个局部时刻。例如， 日，

或者一天中的时间，甚至是一周中的某天。Joda 设计者使用 ReadablePartial 接口捕捉这种表示局部时间的概念，这是一个不可变的局部时间片段。用于处理这种时间片段的两个有用类分别为 LocalDate 和 LocalTime：

- **LocalDate**：该类封装了一个年/月/日的组合。当地理位置（即时区）变得不重要时，使用它存储日期将非常方便。例如，某个特定对象的 *出生日期* 可能为 *1999 年 4 月 16 日*，但是从技术角度来看，在保存所有业务值的同时不会了解有关此日期的任何其他信息（比如这是一周中的星期几，或者这个人出生地所在的时区）。在这种情况下，应当使用 LocalDate。
  样例应用程序使用 SystemClock 来获取被初始化为系统时间的 LocalDate 的实例：

```
LocalDate localDate = SystemFactory.getClock().getLocalDate();
```

  也可以通过显式地提供所含的每个字段的值来创建 LocalDate：

```
LocalDate localDate = new LocalDate(2009, 9, 6);// September 6, 2009
```

  LocalDate 替代了在早期 Joda 版本中使用的 YearMonthDay。

- **LocalTime**：这个类封装一天中的某个时间，当地理位置不重要的情况下，可以使用这个类来只存储一天当中的某个时间。例如，晚上 11:52 可能是一天当中的一个重要时刻（比如，一个 cron 任务将启动，它将备份文件系统的某个部分），但是这个时间并没有特定于某一天，因此我不需要了解有关这一时刻的其他信息。
  样例应用程序使用 SystemClock 获取被初始化为系统时间的 LocalTime 的一个实例：

```
LocalTime localTime = SystemFactory.getClock().getLocalTime();
```

  也可以通过显式地提供所含的每个字段的值来创建 LocalTime：

```
LocalTime localTime = new LocalTime(13, 30, 26, 0);// 1:30:26PM
```

时间跨度

了解特定的时刻或是某个局部时间片段将非…够表达一段时间跨度的话，通常也很有用。Joda …个过程。您可以选择用于表示不同跨度的类：

- **Duration**：这个类表示一个绝对的精确跨度，使用毫秒为单位。这个类提供的方法可以用于通过标准的数学转换（比如 1 分钟 = 60 秒，1 天 = 24 小时），将时间跨度转换为标准单位（比如秒、分和小时）。
  您只在以下情况使用 Duration 的实例：您希望转换一个时间跨度，但是您并不关心这个时间跨度在何时发生，或者使用毫秒处理时间跨度比较方便。

- **Period**：这个类表示与 Duration 相同的概念，但是以人们比较熟悉的单位表示，比如年、月、周。
  您可以在以下情况使用 Period：您并不关心这段时期必须在何时发生，或者您更关心检索单个字段的能力，这些字段描述由 Period 封装的时间跨度。

- **Interval**：这个类表示一个特定的时间跨度，将使用一个明确的时刻界定这段时间跨度的范围。Interval 为*半开*区间，这表示由 Interval 封装的时间跨度包括这段时间的起始时刻，但是不包含结束时刻。
  可以在以下情况使用 Interval：需要表示在时间连续区间中以特定的点开始和结束的一段时间跨度。

以 Joda 的方式处理时间

现在，您已经了解了如何创建一些非常有用的 Joda 类，我将向您展示如何使用它们执行日期计算。接着您将了解到 Joda 如何轻松地与 JDK 进行互操作。

日期计算

如果您只是需要对日期/时间信息使用占位符，那么 JDK 完全可以胜任，但是它在日期/时间计算方面的表现十分糟糕，而这正是 Joda 的长处。我将向您展示一些简单的例子。

假设在当前的系统日期下，我希望计算上一 于这个例子，我并不关心一天中的时间，因为我只 清单 6 所示：

清单 6. 使用 Joda 计算日期

```
LocalDate now = SystemFactory.getClock().getLocalDate();
LocalDate lastDayOfPreviousMonth =\
  now.minusMonths(1).dayOfMonth().withMaximumValue();
```

您可能对清单 6 中的 dayOfMonth() 调用感兴趣。这在 Joda 中被称为*属性（property）*。它相当于 Java 对象的属性。属性是根据所表示的常见结构命名的，并且它被用于访问这个结构，用于完成计算目的。属性是实现 Joda 计算威力的关键。您目前所见到的所有 4 个 Joda 类都具有这样的属性。一些例子包括：

- yearOfCentury
- dayOfYear
- monthOfYear
- dayOfMonth
- dayOfWeek

我将详细介绍清单 6 中的示例，以向您展示整个计算过程。首先，我从当前月份减去一个月，得到 “上一个月”。接着，我要求获得 dayOfMonth 的最大值，它使我得到这个月的最后一天。注意，这些调用被连接到一起（注意 Joda ReadableInstant 子类是不可变的），这样您只需要捕捉调用链中最后一个方法的结果，从而获得整个计算的结果。

当计算的中间结果对我不重要时，我经常会使用这种计算模式。（我以相同的方式使用 JDK 的 BigDecimal）。假设您希望获得任何一年中的第 11 月的第一个星期二的日期，而这天必须是在这个月的第一个星期一之后。清单 7 展示了如何完成这个计算：

清单 7. 计算 11 月中第一个星期一之后的第一个星期二

```
LocalDate now = SystemFactory.getClock().getLocalDate();
LocalDate electionDate = now.monthOfYear()
 .setCopy(11)        // November
 .dayOfMonth()       // Access Day Of Month Property
 .withMinimumValue() // Get its minimum value
 .plusDays(6)        // Add 6 days
 .dayOfWeek()        // Access Day Of Week Property
 .setCopy("Monday")  // Set to Monday (it will round d
 .plusDays(1);       // Gives us Tuesday
```

清单 7 的注释帮助您了解代码如何获得结果。.setCopy("Monday") 是整个计算的关键。不管中间 LocalDate

dayOfWeek 属性设置为 Monday 总是能够四舍五入，这样的话，在每月的开始再加上 6 天就能够让您得到第一个星期一。再加上一天就得到第一个星期二。Joda 使得执行此类计算变得非常容易。

下面是其他一些因为使用 Joda 而变得超级简单的计算：

以下代码计算从现在开始经过两个星期之后的日期：

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime then = now.plusWeeks(2);
```

您可以以这种方式计算从明天起 90 天以后的日期：

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime tomorrow = now.plusDays(1);
DateTime then = tomorrow.plusDays(90);
```

（是的，我也可以向 now 加 91 天，那又如何呢？）

下面是计算从现在起 156 秒之后的时间：

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime then = now.plusSeconds(156);
```

下面的代码将计算五年后的第二个月的最后一天：

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime then = now.minusYears(5) // five years ago
               .monthOfYear()     // get monthOfYear property
               .setCopy(2)        // set it to February
               .dayOfMonth()      // get dayOfMonth property
               .withMaximumValue();// the last day of the month
```

这样的例子实在太多了，我向您已经知道了如何计算。尝试操作一下样例应用程序，亲自体验一下使用 Joda 计算任何日期是多么有趣。

JDK 互操作性

我的许多代码都使用了 JDK Date 和 Calendar 类。但是幸亏有 Joda，我可以执行任何必要的日期算法，然后再转换回 JDK 类。这将两者的优点集中到一起。您在本文中看到的所有 Joda 类都可以从 JDK Calendar 或 Date 创建，正如您在 创建 Joda-Time 对象 中看到的那样。出于同样的原因，可以从您所见过的任何 Joda 类创建 JDK Calendar 或 Date。

清单 8 展示了从 Joda ReadableInstant 多么简单：

清单 8. 从 Joda DateTime 类创建 JDK 类

```
DateTime dateTime = SystemFactory.getClock().getDateTime();
Calendar calendar = dateTime.toCalendar(Locale.getDefault());
Date date = dateTime.toDate();
DateMidnight dateMidnight = SystemFactory.getClock()
  .getDateMidnight();
date = dateMidnight.toDate();
```

对于 ReadablePartial 子类，您还需要经过额外一步，如清单 9 所示：

清单 9. 创建表示 LocalDate 的 Date 对象

```
LocalDate localDate = SystemFactory.getClock().getLocalDate();
Date date = localDate.toDateMidnight().toDate();
```

要创建 Date 对象，它表示从清单 9 所示的 SystemClock 中获得的 LocalDate，您必须首先将它转换为一个 DateMidnight 对象，然后只需要将 DateMidnight 对象作为 Date。（当然，产生的 Date 对象将把它自己的时间部分设置为午夜时刻）。

JDK 互操作性被内置到 Joda API 中，因此您无需全部替换自己的接口，如果它们被绑定到 JDK 的话。比如，您可以使用 Joda 完成复杂的部分，然后使用 JDK 处理接口。

以 Joda 方式格式化时间

使用 JDK 格式化日期以实现打印是完全可以的，但是我始终认为它应该更简单一些。这是 Joda 设计者进行了改进的另一个特性。要格式化一个 Joda 对象，调用它的 toString() 方法，并且如果您愿意的话，传递一个标准的 ISO-8601 或一个 JDK 兼容的控制字符串，以告诉 JDK 如何执行格式化。不需要创建单独的 SimpleDateFormat 对象（但是 Joda 的确为那些喜欢自找麻烦的人提供了一个 DateTimeFormatter 类）。调用 Joda 对象的 toString() 方法，仅此而已。我将展示一些例子。

清单 10 使用了 ISODateTimeFormat 的静态方法：

清单 10. 使用 ISO-8601

```
DateTime dateTime = SystemFactory.getClock().getDateTi
dateTime.toString(ISODateTimeFormat.basicDateTime());
dateTime.toString(ISODateTimeFormat.basicDateTimeNoMil
dateTime.toString(ISODateTimeFormat.basicOrdinalDateTi
dateTime.toString(ISODateTimeFormat.basicWeekDateTime(
```

清单 10 中的四个 toString() 调用分别创

```
20090906T080000.000-0500
20090906T080000-0500
2009249T080000.000-0500
2009W367T080000.000-0500
```

描述	名字	大小

您也可以传递与 SimpleDateFormat JDK 兼容的格式字符串，如清单 11 所示：

清单 11. 传递 SimpleDateFormat 字符串

```
DateTime dateTime = SystemFactory.getClock().getDateTime();
dateTime.toString("MM/dd/yyyy hh:mm:ss.SSSa");
dateTime.toString("dd-MM-yyyy HH:mm:ss");
dateTime.toString("EEEE dd MMMM, yyyy HH:mm:ssa");
dateTime.toString("MM/dd/yyyy HH:mm ZZZZ");
dateTime.toString("MM/dd/yyyy HH:mm Z");

09/06/2009 02:30:00.000PM
06-Sep-2009 14:30:00
Sunday 06 September, 2009 14:30:00PM
09/06/2009 14:30 America/Chicago
09/06/2009 14:30 -0500
```

查看 Javadoc 中有关 joda.time.format.DateTimeFormat 的内容，获得与 JDK SimpleDateFormat 兼容的格式字符串的更多信息，并且可以将其传递给 Joda 对象的 toString() 方法。

结束语

谈到日期处理，Joda 是一种令人惊奇的高效工具。无论您是计算日期、打印日期，或是解析日期，Joda 都将是工具箱中的便捷工具。在本文中，我首先介绍了 Joda，它可以作为 JDK 日期/时间库的替代选择。然后介绍了一些 Joda 概念，以及如何使用 Joda 执行日期计算和格式化。

Joda-Time 衍生了一些相关的项目，您可能会发现这些项目很有用。现在出现了一个针对 Grails Web 开发框架的 Joda-Time 插件。joda-time-jpox 项目的目标就是添加一些必需的映射，以使用 DataNucleus 持久化引擎持久化 Joda-Time 对象。并且，一个针对 Google Web Toolkit（也称为 Goda-Time）的 Joda-Time 实现目前正在开发当中，但是在撰写本文之际因为许可问题而被暂停。访问 参考资料 获得更多信息。

下载

描述	名字

描述	名字	大小
源代码	j-jodatime.zip	812KB

参考资料

学习

- Joda：在 SourceForge 中访问 Joda 项目，并查阅 Javadoc 获得有关 Joda-Time 库的内容。
- Joda-Time Plugin for Grails：了解面向 Grails 的 Joda-Time 插件。
- joda-time-jpox：了解有关 joda-time-jpox 项目的更多内容。
- Goda-Time：关注针对 Google Web Toolkit 的 Joda-Time 移植项目。
- 技术书店：浏览有关这些主题和其他技术主题的图书。
- developerWorks Java 技术专区：提供了数百篇有关 Java 编程各个方面的文章。

获得产品和技术

- Joda-Time：下载 Joda-Time 库。

讨论

- Joda-Time 邮件列表：订阅 Joda-Time 邮件列表或查看列表归档。
- 加入 My developerWorks 社区。