# 旧版API的问题

## 1、线程安全问题

Date类和Calendar类都是可变的，在多线程环境下容易出现并发问题

## 2、设计缺陷

时区处理复杂

月份从0开始计数（0-11）容易造成混淆

格式化和解析操作繁琐

## 3、使用不便

日期计算困难

API方法命名不直观

缺少常用的日期操作方法

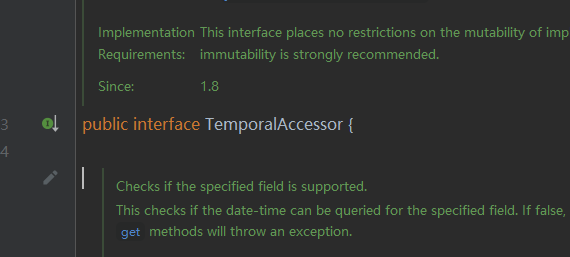
# 接口

## TemporalAccessor

### 基本概念

TemporalAccessor 是 Java 8 引入的日期和时间 API（java.time 包）中的一个接口，它的作用是提供对时间、日期和时间段的访问。该接口主要是用于表示可以读取日期/时间信息的对象，但不提供修改这些信息的方法。

### 源码



### 功能

TemporalAccessor 接口定义了一些方法，让实现该接口的类能够支持读取时间信息，如日期、时间、小时、分钟等。这是一个基础接口，它的主要作用是提供对时间、日期的访问功能，而不是修改功能。

### 方法

#### default int get(TemporalField field)

根据给定的 TemporalField（如 ChronoField.YEAR、ChronoField.MONTH\_OF\_YEAR 等）来获取对应的值。

返回一个整数值，表示时间字段的值。

例如：获取年份、月份、小时等。

#### boolean isSupported(TemporalField field)

判断给定的 TemporalField 是否被当前的 TemporalAccessor 实现所支持。

例如，某些实现可能不支持某些时间字段（比如秒数、时区等）。

#### default ValueRange range(TemporalField field)

返回一个 ValueRange 对象，表示给定 TemporalField 的合法值范围。

例如，获取 MONTH\_OF\_YEAR 的合法值范围是 1 到 12。

### 常见实现

1、LocalDate：表示没有时区的日期，继承自 TemporalAccessor。

2、LocalTime：表示没有日期的时间，继承自 TemporalAccessor。

3、LocalDateTime：表示没有时区的日期时间，继承自 TemporalAccessor。

4、ZonedDateTime：表示带时区的日期时间，继承自 TemporalAccessor。

5、Instant：表示时间戳，继承自 TemporalAccessor。

### 典型用途

TemporalAccessor 是 Temporal 接口的超接口，而 Temporal 提供了时间计算、修改的能力。TemporalAccessor 更多是作为读取和访问日期/时间的接口，允许开发者获取和查询时间字段的值。它常与 Temporal 类一起使用，如 LocalDate, LocalDateTime, ZonedDateTime 等对象，通过 TemporalAccessor 的方法可以访问这些对象的不同部分（如年、月、日等）

### 总结

TemporalAccessor 接口的核心作用是提供对时间或日期的访问功能，而不是修改功能。它为 Temporal 类及其实现提供了通用的时间信息读取方式，使得在处理时间和日期时更具灵活性和扩展性。

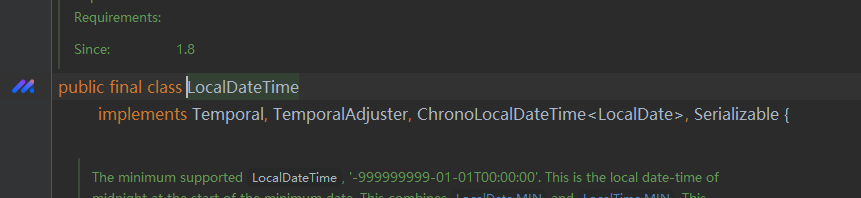
# 类

## LocalDateTime

### 基本概念

LocalDateTime是Java 8中的一个时间类，它可以表示日期和时间，不含时区信息。

### 源码



### 方法

#### public static LocalDateTime now()

获取当前日期和时间

#### public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)

获取指定的日期和时间

#### public int getDayOfMonth()

获取一个月中的天，1-31

#### public DayOfWeek getDayOfWeek()

获取一周中的某天

#### public int getHour()

获取小时数

#### public int getMinute()

获取分钟数

#### public int getSecond()

获取秒，0-59

### 代码示例

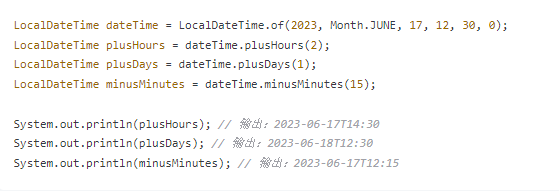
#### 1、获取当前的日期和时间



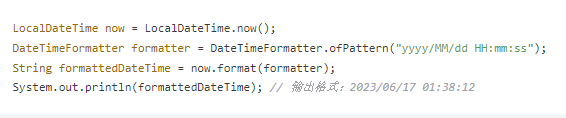
#### 2、创建指定的日期和时间



#### 3、对日期时间对象进行加减操作



#### 4、使用格式化字符串将LocalDateTime对象转换成指定格式的字符串

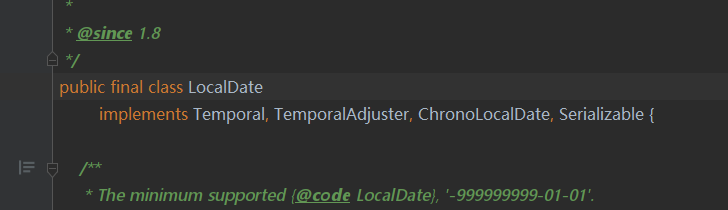


## LocalDate

### 基本概念

LocalDate是Java 8中的一个日期类，它可以表示日期，如年月日等。

### 源码



### 方法

#### public static LocalDate now()

获取当前的日期

#### public static LocalDate parse(CharSequence text, DateTimeFormatter formatter)

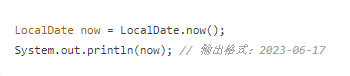
#### public int getYear()

#### public int getMonthValue()

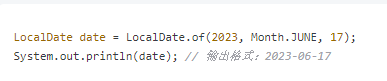
#### public static LocalDate of(int year, int month, int dayOfMonth)

### 示例

#### 1、获取当前的日期



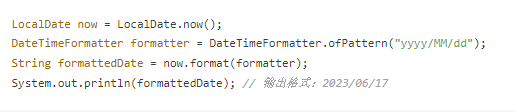
#### 2、创建指定的日期



#### 3、对LocalDate对象进行操作



#### 4、使用格式化字符串将LocalDate对象转换成指定格式的字符串



## LocalTime

### 基本概念

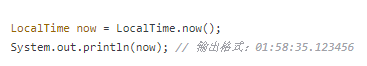
LocalTime是Java 8中的一个时间类，它可以表示时间，如小时、分钟、秒、毫秒等。

### 方法

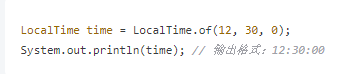
#### public static LocalTime now()

### 示例

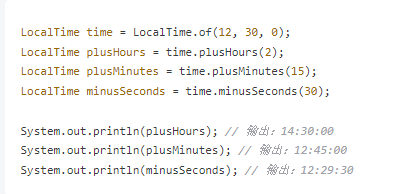
#### 1、获取当前的本地时间



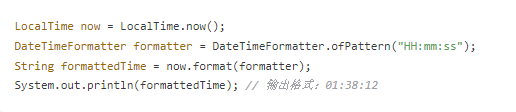
#### 2、创建指定的时间



#### 3、对LocalTime对象进行各种操作



#### 4、使用格式化字符串将LocalTime对象转换成指定格式的字符串

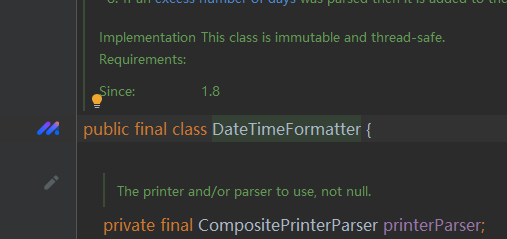


## DateTimeFormatter

### 基本概念

Java 8中的日期时间格式化类，它可以将日期时间对象格式化成指定格式的字符串，或将字符串解析成日期时间对象。

### 源码



### 方法

#### public String format(TemporalAccessor temporal)

格式化日期时间为字符串

#### public static DateTimeFormatter ofPattern(String pattern)

创建指定格式的DateTimeFormatter对象

#### public TemporalAccessor parse(CharSequence text)

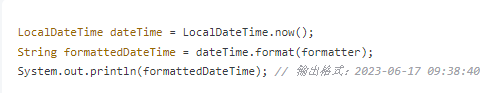
将字符串解析为日期时间

### 示例

#### 1、创建一个格式化字符串的DateTimeFormatter对象



#### 2、将日期时间对象格式化成指定格式的字符串



#### 3、将字符串解析成日期时间对象



## YearMonth

### 基本概念

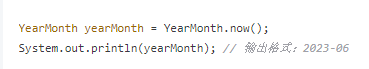
Java 8中表示年月的类，它可以用于计算和处理年月相关的操作

### 方法

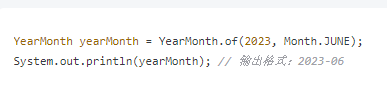
#### public String format(DateTimeFormatter formatter)

### 示例

#### 1、获取当前的年、月



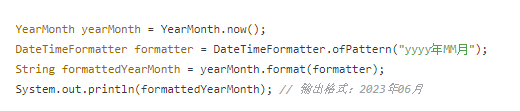
#### 2、创建指定的年月



#### 3、对YearMonth对象进行各种操作



#### 4、使用格式化字符串将YearMonth对象转换成指定格式的字符串



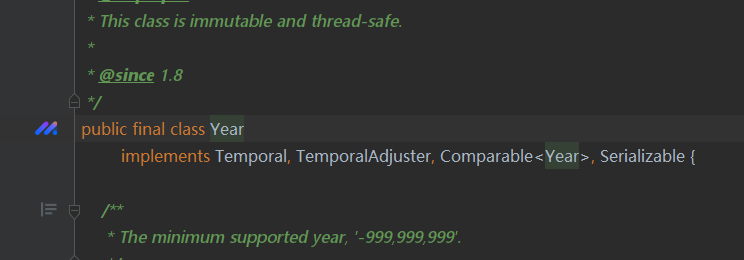
## OffsetDateTime

### 基本概念

## Year

### 基本概念

### 源码



## ZoneId

## ZonedDateTime

## Instant

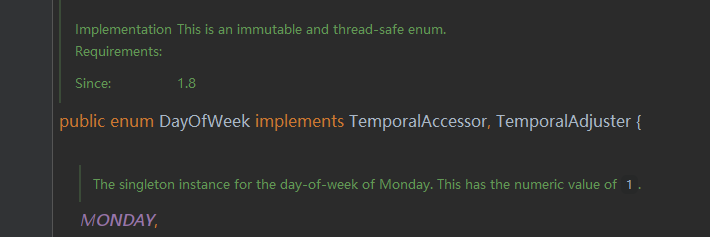
# 枚举

## DayOfWeek

### 基本概念

DayOfWeek 是 Java 8 引入的 java.time 包中的一个枚举类，表示一周的七天。它是 java.time 中用于处理日期的关键类之一，能够帮助我们进行与星期相关的操作，比如确定今天是星期几、获取下一个星期几、比较日期等。

### 源码



### 特性

**1、每个枚举值表示一周的某一天**  
枚举中的每个常量都代表一周中的一天。MONDAY 表示星期一，TUESDAY 表示星期二，以此类推。

**2、从1开始的数字表示**  
DayOfWeek 枚举从 MONDAY 开始，MONDAY 的值是 1，SUNDAY 的值是 7。这是因为在 ISO-8601 标准中，星期一被认为是每周的第一天。

### 方法

#### public int get(TemporalField field)

获取指定字段的值

#### public int getValue()

返回表示星期几的数字。MONDAY 返回 1，TUESDAY 返回 2，依此类推。

#### public static DayOfWeek from(TemporalAccessor temporal)

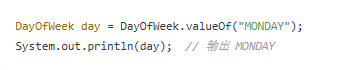
根据给定的日期或时间来获取对应的 DayOfWeek

#### public DayOfWeek plus(long days)

通过 plus 和 minus 方法可以得到下一个或上一个星期几。它们的返回值是一个新的 DayOfWeek 枚举常量。

#### valueOf(String name)

通过名称获取枚举常量。valueOf 是 Enum 类中的方法，可以通过枚举值的名称获取相应的枚举常量。



### 代码示例

假设我们需要根据给定的日期计算出当前日期是星期几，并根据星期几计算出下一个星期几的日期。可以使用以下代码：



### 总结

1、DayOfWeek 枚举提供了一些非常有用的方法，帮助我们对日期进行星期相关的计算。

2、该枚举支持顺序比较、加减天数、获取星期几的数字表示等操作，适合用于处理与星期几相关的逻辑。

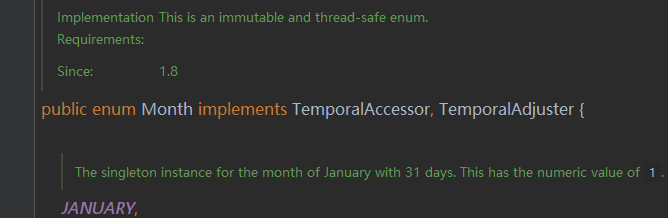
3、由于 DayOfWeek 是枚举类型，因此它是类型安全的，不会出现使用无效或错误星期几的情况。

## Month

### 基本概念

Month 是 Java 8 中引入的 java.time 包中的一个枚举类，代表一年的12个月份。它属于 java.time 包，主要用于处理与月份相关的日期操作。每个枚举常量对应一个月，从 JANUARY（1月）到 DECEMBER（12月）。它提供了很多方法，帮助开发者在日期处理中更加方便和高效。

### 源码



### 特性

1、枚举常量  
每个常量代表一年中的一个月，按顺序从 JANUARY（1月）到 DECEMBER（12月）。

2、数字表示  
Month 枚举常量使用1到12的数字来表示。例如，JANUARY 对应数字 1，DECEMBER 对应数字 12。

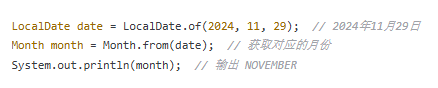
### 方法

#### public int getValue()

返回当前月份的数字表示，JANUARY 返回 1，DECEMBER 返回 12。

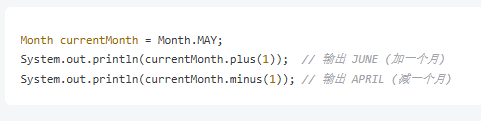
#### public static Month from(TemporalAccessor temporal)

根据给定的日期获取对应的月份



#### public Month plus(long months)

允许你从当前月份加上指定的月份数，返回新的 Month 枚举值

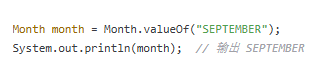


#### public Month minus(long months)

允许你从当前月份减去指定的月份数，返回新的 Month 枚举值

#### valueOf(String name)

根据字符串获取对应的 Month 枚举常量



#### public int length(boolean leapYear)

返回当前月份的天数。对于 FEBRUARY（2月），此方法会考虑是否为闰年。如果是闰年，FEBRUARY 会有 29 天，否则为 28 天。

#### public int firstDayOfYear(boolean leapYear)

对应于本月第一天的一年中的第几天，从1到336

### 代码示例

假设我们需要获取当前日期的月份，并计算从当前月份开始的下一个季度的第一个月。



### 总结

1、Month 枚举提供了很多方法，帮助开发者方便地操作和计算月份。

2、它可以进行月份加减、判断前后关系、获取月份的天数等操作。

3、Month 枚举非常适合在需要进行日期计算的场景中使用，结合 LocalDate 和其他日期时间类一起，能够简化日期的处理和计算。

# 最佳实践建议

## 1、优先使用不可变类型

使用LocalDate、LocalTime、LocalDateTime等不可变类型

避免使用旧的Date类

## 2、合理选择日期时间类型

纯日期使用LocalDate

纯时间使用LocalTime

日期时间都需要使用LocalDateTime

需要时区信息使用ZonedDateTime

## 3、使用标准格式化器

优先使用预定义的DateTimeFormatter

自定义格式化器时注意线程安全

## 4、注意性能优化

DateTimeFormatter实例化开销较大，建议静态定义

批量操作时注意避免重复创建对象