Java 8 新特性

> 主题：Java 8 新特性
>
> 文档编写：沈扬凯
>
> 分享：沈扬凯
>
> 部门：数据支撑平台开发小组

Java 8 的发展

JDK 5

自动装箱与拆箱

JDK1.5为每一个基本数据类型定义了一个封装类。使java中的基本数据类型也有自己的对象

```
int -->Integer
double --> Double
long --> Long
char --> Character
float --> Float
boolean --> Boolean
short --> Short
byte -- > Byte
```

- 自动装包：将基本类型转换成为对象，例如：`int --> Integer`
- 自动拆包：将对象转换成为基本数据类型，例如：`Integer --> int`

对于 JDK1.5 之前集合总不能存放基本数据类型的问题，现在也能够解决。

枚举

枚举是 JDK1.5 推出的一个比较重要的特性。其关键字为 `enum` 例如：定义代表交通灯的枚举

```
public enum MyEnum{
    RED,GREEN,YELLOW
}
```

静态导入

- 优点：使用静态导入可以使被导入类的所有静态变量和静态方法在当前类直接可见，使用这些静态成员无需再给出他们的类名。
- 缺点：过度使用会降低代码的可读性

可变参数

在JDK1.5以前，当我们要为一个方法传递多个类型相同的参数时， 我们有两种方法解决

1. 直接传递一个数组过去
2. 有多少个参数就传递多少个参数。

例如：

```
public void printColor(String red,String green,String yellow){
}
```

或者

```
public void printColor(String[] colors){

}
```

这样编写方法参数虽然能够实现我们想要的效果，但是，这样是不是有点麻烦呢？ 再者，如果参数个数不确定，我们怎么办呢？Java JDK1.5为我们提供的可变参数就能够完美的解决这个问题.

例如：

```
public void printColor(String... colors){

}
```

如果参数的类型相同，那么可以使用 `类型+三个点` ，后面跟一个参数名称的形式。 这样的好处就是，只要参数类型相同，无论传递几个参数都没有限制 注意：可变参数必须是参数列表的最后一项（该特性对对象和基本数据类型都适用）

泛型

```
//给集合指定存入类型，上面这个集合在存入数据的时候必须存入String类型的数据，否则编译器会报错
List<String> strs = new ArrayList<String>();
```

“泛型” 意味着编写的代码可以被不同类型的对象所重用。 可见泛型的提出是为了编写重用性更好的代码。 泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

比如常见的集合类 `LinkedList` ，其实现的接口名后有个特殊的部分 `<>` ，而且它的成员的类型 Link 也包含一个 `<>` ，这个符号的就是类型参数， 它使得在运行中，创建一个 LinkedList 时可以传入不同的类型，比如 `new LinkedList` ，这样它的成员存放的类型也是 `String` 。

For-Each循环

例如上面这个集合我们可以通过for-each遍历，这样更加简单清晰。

```
for(String s : strs){
     System.out.println(s);
}
```

> 注意：使用for-each遍历集合时，要遍历的集合必须实现了Iterator接口

线程并发库

线程并发库是 Java1.5 提出的关于多线程处理的高级功能，所在包： `java.util.concurrent` 包括

1. 线程互斥工具类： `Lock` ， `ReadWriteLock`
2. 线程通信： `Condition`
3. 线程池： `ExecutorService`
4. 同步队列： `ArrayBlockingQueue`
5. 同步集合： `ConcurrentHashMap` ， `CopyOnWriteArrayList`
6. 线程同步工具： `Semaphore`

JDK 6

Desktop类和SystemTray类

前者可以用来打开系统默认浏览器浏览指定的URL，打开系统默认邮件客户端给指定的邮箱发邮件，用默认应用程序打开或编辑文件(比如，用记事本打开以 txt 为后缀名的文件)，用系统默认的打印机打印文档；后者可以用来在系统托盘区创建一个托盘程序。

使用Compiler API

现在我们可以用JDK1.6 的Compiler API(JSR 199)去动态编译Java源文件，Compiler API结合反射功能就可以实现动态的产生Java代码并编译执行这些代码，有点动态语言的特征。

这个特性对于某些需要用到动态编译的应用程序相当有用，比如JSP Web Server，当我们手动修改JSP后，是不希望需要重启Web Server才可以看到效果的，这时候我们就可以用Compiler API来实现动态编译JSP文件。

当然，现在的JSP Web Server也是支持JSP热部署的，现在的JSP Web Server通过在运行期间通过Runtime.exec或ProcessBuilder来调用javac来编译代码，这种方式需要我们产生另一个进程去做编译工作，不够优雅而且容易使代码依赖与特定的操作系统；

Compiler API通过一套易用的标准的API提供了更加丰富的方式去做动态编译，而且是跨平台的。

轻量级Http Server API

JDK1.6 提供了一个简单的 Http Server API，据此我们可以构建自己的嵌入式 Http Server，它支持Http和Https协议，提供了HTTP1.1的部分实现，没有被实现的那部分可以通过扩展已有的 Http Server API来实现，程序员必须自己实现 HttpHandler 接口，HttpServer 会调用 `HttpHandler` 实现类的回调方法来处理客户端请求，在这里，我们把一个 Http 请求和它的响应称为一个交换，包装成 `HttpExchange` 类， `HttpServer` 负责将 `HttpExchange` 传给 `HttpHandler` 实现类的回调方法。

用Console开发控制台程序

JDK1.6 中提供了 `java.io.Console` 类专用来访问基于字符的控制台设备。 你的程序如果要与 Windows 下的 cmd 或者 Linux 下的 Terminal 交互，就可以用 `Console` 类代劳。 但我们不总是能得到可用的 Console，一个JVM是否有可用的 Console 依赖于底层平台和 JVM 如何被调用。 如果JVM是在交互式命令行(比如 Windows 的 cmd)中启动的，并且输入输出没有重定向到另外的地方，那么就可以得到一个可用的 Console 实例。

对脚本语言的支持

如：ruby，groovy，javascript。

JDK 7

数字变量对下滑线的支持

JDK1.7可以在数值类型的变量里添加下滑线。

例如：

```
int num = 1234_5678_9;
float num2 = 222_33F;
long num3 = 123_000_111L;
```

注意，有几个地方是不能添加的：

1. 数字的开头和结尾
2. 小数点前后
3. F或者L前

switch对String的支持

```
String status = "orderState";
switch (status) {
    case "ordercancel":
        System.out.println("订单取消");
        break;
    case "orderSuccess":
        System.out.println("预订成功");
        break;
    default:
        System.out.println("状态未知");
}
```

try-with-resource

- `try-with-resources` 是一个定义了一个或多个资源的 try 声明，这个资源是指程序处理完它之后需要关闭它的对象。
- `try-with-resources` 确保每一个资源在处理完成后都会被关闭。

可以使用try-with-resources的资源有： 任何实现了 `java.lang.AutoCloseable` 接口 `java.io.Closeable` 接口的对象。

例如：

```
public static String readFirstLineFromFile(String path) throws IOException {

    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

在 java 7 以及以后的版本里， `BufferedReader` 实现了 `java.lang.AutoCloseable` 接口。 由于 `BufferedReader` 定义在 `try-with-resources` 声明里，无论 `try` 语句正常还是异常的结束， 它都会自动的关掉。而在 java7 以前，你需要使用 `finally` 块来关掉这个对象。

捕获多种异常并用改进后的类型检查来重新抛出异常

```
public static void first(){
    try {
        BufferedReader reader = new BufferedReader(new FileReader(""));
        Connection con = null;
        Statement stmt = con.createStatement();
    } catch (IOException | SQLException e) {
        //捕获多个异常，e就是final类型的
        e.printStackTrace();
    }
}
```

优点：用一个 `catch` 处理多个异常，比用多个 `catch` 每个处理一个异常生成的字节码要更小更高效。

创建泛型时类型推断

只要编译器可以从上下文中推断出类型参数，你就可以用一对空着的尖括号 `<>` 来代替泛型参数。这对括号私下被称为菱形(diamond)。 在Java SE 7之前，你声明泛型对象时要这样

```
List<String> list = new ArrayList<String>();
```

而在Java SE7以后，你可以这样

```
List<String> list = new ArrayList<>();
```

因为编译器可以从前面(List)推断出推断出类型参数，所以后面的 `ArrayList` 之后可以不用写泛型参数了，只用一对空着的尖括号就行。 当然，你必须带着菱形 `<>`，否则会有警告的。 Java SE7 只支持有限的类型推断：只有构造器的参数化类型在上下文中被显著的声明了，你才可以使用类型推断，否则不行。

```
List<String> list = new ArrayList<>();l
list.add("A");
//这个不行
list.addAll(new ArrayList<>());
// 这个可以
List<? extends String> list2 = new ArrayList<>();
list.addAll(list2);
```

JDK 8

Lambda表达式和函数式接口

Lambda表达式（也称为闭包）是Java 8中最大和最令人期待的语言改变。它允许我们将函数当成参数传递给某个方法，或者把代码本身当作数据处理：函数式开发者非常熟悉这些概念。很多JVM平台上的语言（Groovy、Scala等）从诞生之日就支持Lambda表达式，但是Java开发者没有选择，只能使用匿名内部类代替Lambda表达式。 Lambda的设计耗费了很多时间和很大的社区力量，最终找到一种折中的实现方案，可以实现简洁而紧凑的语言结构。最简单的Lambda表达式可由逗号分隔的参数列表、->符号和语句块组成。

Lambda的设计者们为了让现有的功能与Lambda表达式良好兼容，考虑了很多方法，于是产生了函数接口这个概念。函数接口指的是只有一个函数的接口，这样的接口可以隐式转换为Lambda表达式。java.lang.Runnable和java.util.concurrent.Callable是函数式接口的最佳例子。在实践中，函数式接口非常脆弱：只要某个开发者在该接口中添加一个函数，则该接口就不再是函数式接口进而导致编译失败。为了克服这种代码层面的脆弱性，并显式说明某个接口是函数式接口，Java 8 提供了一个特殊的注解@FunctionalInterface（Java 库中的所有相关接口都已经带有这个注解了）。

接口的默认方法和静态方法

Java 8使用两个新概念扩展了接口的含义：默认方法和静态方法。默认方法使得接口有点类似traits，不过要实现的目标不一样。默认方法使得开发者可以在 不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不强制那些实现了该接口的类也同时实现这个新加的方法。 默认方法和抽象方法之间的区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方法会被接口的实现类继承或者覆写。 由于JVM上的默认方法的实现在字节码层面提供了支持，因此效率非常高。默认方法允许在不打破现有继承体系的基础上改进接口。该特性在官方库中的应用是：给java.util.Collection接口添加新方法，如stream()、parallelStream()、forEach()和removeIf()等等。 尽管默认方法有这么多好处，但在实际开发中应该谨慎使用：在复杂的继承体系中，默认方法可能引起歧义和编译错误。如果想了解更多细节，可以参考官方文档。

更好的类型推断

Java 8 编译器在类型推断方面有很大的提升，在很多场景下编译器可以推导出某个参数的数据类型，从而使得代码更为简洁。

参数 `Value.defaultValue()` 的类型由编译器推导得出，不需要显式指明。在Java 7中这段代码会有编译错误，除非使用 `Value.<String>defaultValue()` 。

Optional

Java应用中最常见的bug就是空指针异常。在Java 8之前，Google Guava引入了 `Optionals` 类来解决 `NullPointerException` ，从而避免源码被各种 `null` 检查污染，以便开发者写出更加整洁的代码。Java 8也将Optional加入了官方库。 `Optional` 仅仅是一个容易存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检查，可以参考Java 8官方文档了解更多细节。

如果Optional实例持有一个非空值，则 `isPresent()` 方法返回true，否则返回false； `orElseGet()` 方法，Optional实例持有null，则可以接受一个lambda表达式生成的默认值；map()方法可以将现有的 `Optional` 实例的值转换成新的值；orElse()方法与orElseGet()方法类似，但是在持有null的时候返回传入的默认值。

Stream

新增的Stream API（java.util.stream）将生成环境的函数式编程引入了Java库中。这是目前为止最大的一次对Java库的完善，以便开发者能够写出更加有效、更加简洁和紧凑的代码。

Task 类有一个分数（或伪复杂度）的概念，另外还有两种状态：OPEN 或者 CLOSED。现在假设有一个task集合，首先看一个问题：在这个task集合中一共有多少个OPEN状态的点?

在Java 8之前，要解决这个问题，则需要使用foreach循环遍历task集合；但是在Java 8中可以利用steams解决：包括一系列元素的列表，并且支持顺序和并行处理。

```
final Collection<Task> tasks = Arrays.asList(
        new Task(Status.OPEN, 5),
        new Task(Status.OPEN, 13),
        new Task(Status.CLOSED, 8)
);

// 使用sum()计算所有 OPEN 任务
final long totalPointsOfOpenTasks = tasks
        .stream()
        .filter(task -> task.getStatus() == Status.OPEN)
```

```
        .mapToInt(Task::getPoints)
        .sum();

System.out.println("Total points: " + totalPointsOfOpenTasks);
```

首先，tasks集合被转换成steam表示；其次，在steam上的filter操作会过滤掉所有CLOSED的task；第三，mapToInt操作基于每个task实例的`Task::getPoints`方法将task流转换成Integer集合；最后，通过sum方法计算总和，得出最后的结果。

新的日期时间 API

Java 8引入了新的Date-Time API(JSR 310)来改进时间、日期的处理。时间和日期的管理一直是最令Java开发者痛苦的问题。java.util.Date 和后来的 java.util.Calendar 一直没有解决这个问题（甚至令开发者更加迷茫）。因为上面这些原因，诞生了第三方库Joda-Time，可以替代Java的时间管理API。

Java 8中新的时间和日期管理API深受Joda-Time影响，并吸收了很多Joda-Time的精华。

第一，新的java.time包包含了所有关于日期、时间、时区、Instant（跟日期类似但是精确到纳秒）、duration（持续时间）和时钟操作的类。新设计的API认真考虑了这些类的不变性（从java.util.Calendar吸取的教训），如果某个实例需要修改，则返回一个新的对象。

第二，关注下LocalDate和LocalTime类。LocalDate仅仅包含ISO-8601日历系统中的日期部分；LocalTime则仅仅包含该日历系统中的时间部分。这两个类的对象都可以使用Clock对象构建得到。

第三，LocalDateTime类包含了LocalDate和LocalTime的信息，但是不包含ISO-8601日历系统中的时区信息。这里有一些关于LocalDate和LocalTime的例子：

如果你需要特定时区的data/time信息，则可以使用ZoneDateTime，它保存有ISO-8601日期系统的日期和时间，而且有时区信息。

Nashorn JavaScript引擎

Java 8提供了新的Nashorn JavaScript引擎，使得我们可以在JVM上开发和运行JS应用。 Nashorn JavaScript引擎是javax.script.ScriptEngine的另一个实现版本，这类Script引擎遵循相同的规则，允许Java和JavaScript交互使用。

Base64

对 Base64 编码的支持已经被加入到Java 8官方库中，这样不需要使用第三方库就可以进行Base64编码。

Lambda 表达式（★★）

PS：个人理解类似前端 ES6 的箭头函数

命令式和函数式

命令式编程：命令“机器”如何去做事情(how)，这样不管你想要的是什么(what)，它都会按照你的命令实现。 **声明式编程**：告诉“机器”你想要的是什么(what)，让机器想出如何去做(how)。

什么是函数式编程?

每个人对函数式编程的理解不尽相同。 我的理解是：**在完成一个编程任务时，通过使用不可变的值或函数，对他们进行处理，然后得到另一个值的过程。** 不同的语言社区往往对各自语言中的特性孤芳自赏。现在谈 Java 程序员如何定义函数式编程还为时尚早，但是，这根本不重要！ 我们关心的是如何写出好代码，而不是符合函数式编程风格的代码。

行为参数化

把算法的策略（行为）作为一个参数传递给函数。

lambda 特点

- 匿名：它不像普通的方法那样有一个明确的名称：写得少而想得多！
- 函数：Lambda函数不像方法那样属于某个特定的类。但和方法一样，Lambda有参数列表、函数主体、返回类型，还可能有可以抛出的异常列表。
- 传递：Lambda表达式可以作为参数传递给方法或存储在变量中。
- 简洁：无需像匿名类那样写很多模板代码。

函数描述符

函数式接口的抽象方法的签名基本上就是Lambda表达式的签名，这种抽象方法叫作函数描述符。

函数式接口，类型推断

函数式接口定义且只定义了一个抽象方法，因为抽象方法的签名可以描述Lambda表达式的签名。 函数式接口的抽象方法的签名称为函数描述符。 所以为了应用不同的Lambda表达式，你需要一套能够描述常见函数描述符的函数式接口。

Lambdas及函数式接口的例子

使用案例	Lambda 的例子	对应的函数式接口
布尔表达式	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
创建对象	`() -> new Project()`	`Supplier<Project>`
消费一个对象	`(Project p) -> System.out.println(p.getStars())`	`Consumer<Project>`
从一个对象中选择/提取	`(int a, int b) -> a * b`	`IntBinaryOperator`
比较两个对象	`(Project p1, Project p2) -> p1.getStars().compareTo(p2.getStars())`	`Comparator<Project>` 或 `BiFunction<Project, Project, Integer>` 或 `ToIntBiFunction<Project, Project>`

Lambda 小结

- lambda 表达式可以理解为一种匿名函数：它没有名称，但有参数列表、函数主体、返回 类型，可能还有一个可以抛出的异常的列表。
- lambda 表达式让你可以简洁地传递代码。
- 只有在接受函数式接口的地方才可以使用 lambda 表达式。

- lambda 表达式允许你直接内联，为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。
- Lambda表达式所需要代表的类型称为目标类型。

方法引用

方法引用让你可以重复使用现有的方法定义，并像Lambda一样传递它们。

方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

方法引用使用一对冒号`::`。

示例：

```
List<Person> result = list.stream()
                    .filter(Person::isStudent) // 就是方法引用
                    .collect(Collectors.toList());
```

函数式接口

函数式接口(Functional Interface)就是**一个有且仅有一个抽象方法**，但是可以有多个非抽象方法的接口。

函数式接口可以被隐式转换为lambda表达式。

函数式接口可以现有的函数友好地支持 lambda。

JDK 1.8之前已有的函数式接口:

- **java.lang.Runnable**
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- **java.util.Comparator**
- java.io.FileFilter
- java.nio.file.PathMatcher
- java.lang.reflect.InvocationHandler
- java.beans.PropertyChangeListener
- java.awt.event.ActionListener
- javax.swing.event.ChangeListener

JDK 1.8 新增加的函数接口：

- java.util.function

`java.util.function` 这个包下包含了很多类，用来支持 Java的 函数式编程，包括 `Predicate<T>`、`Function<T,R>`、`Supplier<T>`、`Consumer<T>` 和 `BinaryOperator<T>`。

`Comparator`、`Predicate` 和 `Function` 等函数式接口都有几个可以用来结合 lambda 表达式的默认方法。

Java 8中的常用函数式接口

函数式接口	函数描述符	原始类型特化
`Predicate<T>`	`T->boolean`	`IntPredicate,LongPredicate, DoublePredicate`
`Consumer<T>`	`T->void`	`IntConsumer,LongConsumer, DoubleConsumer`
`Function<T,R>`	`T->R`	`IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>`
`Supplier<T>`	`()->T`	`BooleanSupplier,IntSupplier, LongSupplier, DoubleSupplier`
`UnaryOperator<T>`	`T->T`	`IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator`
`BinaryOperator<T>`	`(T,T)->T`	`IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator`
`BiPredicate<L,R>`	`(L,R)->boolean`	
`BiConsumer<T,U>`	`(T,U)->void`	`ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>`
`BiFunction<T,U,R>`	`(T,U)->R`	`ToIntBiFunction<T,U>, ToLongBiFunction<T,U>, ToDoubleBiFunction<T,U>`

默认方法

新增了接口的默认方法。

简单说，默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。

我们只需在方法名前面加个default关键字即可实现默认方法。

Stream（★★）

关于流

什么是流?

流是Java8引入的全新概念，它用来处理集合中的数据，暂且可以把它理解为一种高级集合。 众所周知，集合操作非常麻烦，若要对集合进行筛选、投影，需要写大量的代码，而流是以声明的形式操作集合，它就像SQL语句，我们只需告诉流需要对集合进行什么操作，它就会自动进行操作，并将执行结果交给你，无需我们自己手写代码。 因此，流的集合操作对我们来说是透明的，我们只需向流下达命令，它就会自动把我们想要的结果给我们。由于操作过程完全由Java处理，因此它可以根据当前硬件环境选择最优的方法处理，我们也无需编写复杂又容易出错的多线程代码了。

流的特点

1. 只能遍历一次
   我们可以把流想象成一条流水线，流水线的源头是我们的数据源(一个集合)，数据源中的元素依次被输送到流水线上，我们可以在流水线上对元素进行各种操作。
   一旦元素走到了流水线的另一头，那么这些元素就被“消费掉了”，我们无法再对这个流进行操作。当然，我们可以从数据源那里再获得一个新的流重新遍历一遍。
2. 采用内部迭代方式
   若要对集合进行处理，则需我们手写处理代码，这就叫做外部迭代。
   而要对流进行处理，我们只需告诉流我们需要什么结果，处理过程由流自行完成，这就称为内部迭代。

流的操作种类

流的操作分为两种，分别为中间操作和终端操作。

1. 中间操作
   当数据源中的数据上了流水线后，这个过程对数据进行的所有操作都称为“中间操作”。
   中间操作仍然会返回一个流对象，因此多个中间操作可以串连起来形成一个流水线。
2. 终端操作
   当所有的中间操作完成后，若要将数据从流水线上拿下来，则需要执行终端操作。
   终端操作将返回一个执行结果，这就是你想要的数据。

流的操作过程

使用流一共需要三步：

1. 准备一个数据源
2. 执行中间操作
   中间操作可以有多个，它们可以串连起来形成流水线。
3. 执行终端操作
   执行终端操作后本次流结束，你将获得一个执行结果。

使用流

创建流

在使用流之前，首先需要拥有一个数据源，并通过StreamAPI提供的一些方法获取该数据源的流对象。数据源可以有多种形式：

1. 集合

这种数据源较为常用，通过stream()方法即可获取流对象：

```
List<Person> list = new ArrayList<Person>();
Stream<Person> stream = list.stream();
```

2. 数组

通过Arrays类提供的静态函数stream()获取数组的流对象：

```
String[] names = {"chaimm","peter","john"};
Stream<String> stream = Arrays.stream(names);
```

3. 值

直接将几个值变成流对象：

```
Stream<String> stream = Stream.of("chaimm","peter","john");
```

4. 文件

```
try(Stream lines = Files.lines(Paths.get("文件路径名"),Charset.defaultCharset()))
{
    //可对lines做一些操作
}catch(IOException e){
}
```

> PS：Java7简化了IO操作，把打开IO操作放在try后的括号中即可省略关闭IO的代码。

5. iterator

创建无限流

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```

筛选 filter

filter 函数接收一个Lambda表达式作为参数，该表达式返回boolean，在执行过程中，流将元素逐一输送给filter，并筛选出执行结果为true的元素。 如，筛选出所有学生：

```
List<Person> result = list.stream()
                    .filter(Person::isStudent)
                    .collect(Collectors.toList());
```

去重distinct

去掉重复的结果：

```
List<Person> result = list.stream()
                    .distinct()
                    .collect(Collectors.toList());
```

> PS：对象去重的时候，需要注意重写 equals 和 hashcode 方法

截取

截取流的前N个元素：

```
List<Person> result = list.stream()
                    .limit(3)
                    .collect(Collectors.toList());
```

跳过

跳过流的前n个元素：

```
List<Person> result = list.stream()
                    .skip(3)
                    .collect(Collectors.toList());
```

映射

对流中的每个元素执行一个函数，使得元素转换成另一种类型输出。流会将每一个元素输送给map函数，并执行map中的Lambda表达式，最后将执行结果存入一个新的流中。 如，获取每个人的姓名(实则是将Perosn类型转换成String类型)：

```
List<Person> result = list.stream()
                    .map(Person::getName)
                    .collect(Collectors.toList());
```

合并多个流

例：列出List中各不相同的单词，List集合如下：

```
List<String> list = new ArrayList<String>();
list.add("I am a boy");
list.add("I love the girl");
list.add("But the girl loves another girl");
```

思路如下：

首先将list变成流：

```
list.stream();
```

按空格分词：

```
list.stream()
            .map(line->line.split(" "));
```

分完词之后，每个元素变成了一个String[]数组。

将每个 `String[]` 变成流：

```
list.stream()
            .map(line->line.split(" "))
            .map(Arrays::stream)
```

此时一个大流里面包含了一个个小流，我们需要将这些小流合并成一个流。

将小流合并成一个大流：用 `flatMap` 替换刚才的 map

```
list.stream()
    .map(line->line.split(" "))
    .flatMap(Arrays::stream)
```

去重

```
list.stream()
    .map(line->line.split(" "))
    .flatMap(Arrays::stream)
    .distinct()
    .collect(Collectors.toList());
```

是否匹配任一元素：anyMatch

anyMatch用于判断流中是否存在至少一个元素满足指定的条件，这个判断条件通过Lambda表达式传递给anyMatch，执行结果为boolean类型。 如，判断list中是否有学生：

```
boolean result = list.stream()
            .anyMatch(Person::isStudent);
```

是否匹配所有元素：allMatch

allMatch用于判断流中的所有元素是否都满足指定条件，这个判断条件通过Lambda表达式传递给anyMatch，执行结果为boolean类型。 如，判断是否所有人都是学生：

```
boolean result = list.stream()
            .allMatch(Person::isStudent);
```

是否未匹配所有元素：noneMatch

noneMatch与allMatch恰恰相反，它用于判断流中的所有元素是否都不满足指定条件：

```
boolean result = list.stream()
            .noneMatch(Person::isStudent);
```

获取任一元素findAny

findAny能够从流中随便选一个元素出来，它返回一个Optional类型的元素。

```
Optional<Person> person = list.stream().findAny();
```

获取第一个元素findFirst

```
Optional<Person> person = list.stream().findFirst();
```

归约

归约是将集合中的所有元素经过指定运算，折叠成一个元素输出，如：求最值、平均数等，这些操作都是将一个集合的元素折叠成一个元素输出。

在流中，reduce函数能实现归约。 reduce函数接收两个参数：

1. 初始值
2. 进行归约操作的Lambda表达式

元素求和：自定义Lambda表达式实现求和

例：计算所有人的年龄总和

```
int age = list.stream().reduce(0, (person1,person2)-
>person1.getAge()+person2.getAge());
```

1. reduce的第一个参数表示初试值为0；
2. reduce的第二个参数为需要进行的归约操作，它接收一个拥有两个参数的Lambda表达式，reduce会把流中的元素两两输给Lambda表达式，最后将计算出累加之和。

元素求和：使用Integer.sum函数求和

上面的方法中我们自己定义了Lambda表达式实现求和运算，如果当前流的元素为数值类型，那么可以使用Integer提供了sum函数代替自定义的Lambda表达式，如：

```
int age = list.stream().reduce(0, Integer::sum);
```

Integer类还提供了 `min` 、 `max` 等一系列数值操作，当流中元素为数值类型时可以直接使用。

数值流的使用

采用reduce进行数值操作会涉及到基本数值类型和引用数值类型之间的装箱、拆箱操作，因此效率较低。 当流操作为纯数值操作时，使用数值流能获得较高的效率。

将普通流转换成数值流

StreamAPI提供了三种数值流：IntStream、DoubleStream、LongStream，也提供了将普通流转换成数值流的三种方法：mapToInt、mapToDouble、mapToLong。 如，将Person中的age转换成数值流：

```
IntStream stream = list.stream().mapToInt(Person::getAge);
```

数值计算

每种数值流都提供了数值计算函数，如max、min、sum等。如，找出最大的年龄：

```
OptionalInt maxAge = list.stream()
                                .mapToInt(Person::getAge)
                                .max();
```

由于数值流可能为空，并且给空的数值流计算最大值是没有意义的，因此max函数返回`OptionalInt`，它是`Optional`的一个子类，能够判断流是否为空，并对流为空的情况作相应的处理。 此外，`mapToInt`、`mapToDouble`、`mapToLong`进行数值操作后的返回结果分别为：`OptionalInt`、`OptionalDouble`、`OptionalLong`

中间操作和收集操作

操作	类型	返回类型	使用的类型/函数式接口	函数描述符
`filter`	中间	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`distinct`	中间	`Stream<T>`		
`skip`	中间	`Stream<T>`	long	
`map`	中间	`Stream<R>`	`Function<T, R>`	`T -> R`
`flatMap`	中间	`Stream<R>`	`Function<T, Stream<R>>`	`T -> Stream<R>`
`limit`	中间	`Stream<T>`	long	
`sorted`	中间	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`anyMatch`	终端	`boolean`	`Predicate<T>`	`T -> boolean`
`noneMatch`	终端	`boolean`	`Predicate<T>`	`T -> boolean`
`allMatch`	终端	`boolean`	`Predicate<T>`	`T -> boolean`
`findAny`	终端	`Optional<T>`		
`findFirst`	终端	`Optional<T>`		
`forEach`	终端	`void`	`Consumer<T>`	`T -> void`
`collect`	终端	`R`	`Collector<T, A, R>`	
`reduce`	终端	`Optional<T>`	`BinaryOperator<T>`	`(T, T) -> T`
`count`	终端	`long`		

Collector 收集

收集器用来将经过筛选、映射的流进行最后的整理，可以使得最后的结果以不同的形式展现。`collect` 方法即为收集器，它接收 `Collector` 接口的实现作为具体收集器的收集方法。`Collector` 接口提供了很多默认实现的方法，我们可以直接使用它们格式化流的结果；也可以自定义 `Collector` 接口的实现，从而定制自己的收集器。

归约

流由一个个元素组成，归约就是将一个个元素“折叠”成一个值，如求和、求最值、求平均值都是归约操作。

一般性归约

若你需要自定义一个归约操作，那么需要使用 `Collectors.reducing` 函数，该函数接收三个参数：

- 第一个参数为归约的初始值
- 第二个参数为归约操作进行的字段
- 第三个参数为归约操作的过程

汇总

Collectors类专门为汇总提供了一个工厂方法：`Collectors.summingInt`。 它可接受一 个把对象映射为求和所需int的函数，并返回一个收集器；该收集器在传递给普通的 `collect` 方法后即执行我们需要的汇总操作。

分组

数据分组是一种更自然的分割数据操作，分组就是将流中的元素按照指定类别进行划分，类似于SQL语句中的 `GROUPBY`。

多级分组

多级分组可以支持在完成一次分组后，分别对每个小组再进行分组。 使用具有两个参数的 `groupingBy` 重载方法即可实现多级分组。

- 第一个参数：一级分组的条件
- 第二个参数：一个新的 `groupingBy` 函数，该函数包含二级分组的条件

Collectors 类的静态工厂方法

工厂方法	返回类型	用途	示例
`toList`	`List<T>`	把流中所有项目收集到一个 List	`List<Project> projects = projectStream.collect(toList(`
`toSet`	`Set<T>`	把流中所有项目收集到一个 Set，删除重复项	`Set<Project> projects = projectStream.collect(toSet())`
`toCollection`	`Collection<T>`	把流中所有项目收集到给定的供应源创建的集合	`Collection<Project> projects = projectStream.collect(toCollection(), ArrayList::new);`
`counting`	`Long`	计算流中元素的个数	`long howManyProjects = projectStream.collect(counting(`
`summingInt`	`Integer`	对流中项目的一个整数属性求和	`int totalStars = projectStream.collect(summingInt(Project::getStars));`
`averagingInt`	`Double`	计算流中项目 Integer 属性的平均值	`double avgStars = projectStream.collect(averagingInt(Project::getStars))`
`summarizingInt`	`IntSummaryStatistics`	收集关于流中项目 Integer 属性的统计值，例如最大、最小、 总和与平均值	`IntSummaryStatistics projectStatistics = projectStream.collect(summarizingInt(Project::getStars`
`joining`	`String`	连接对流中每个项目调用 toString 方法所生成的字符串	`String shortProject = projectStream.map(Project::getName).collect(joining(",`
`maxBy`	`Optional<T>`	按照给定比较器选出的最大元素的 Optional， 或如果流为空则为 Optional.empty()	`Optional<Project> fattest = projectStream.collect(maxBy(comparingInt(Project::getS`
`minBy`	`Optional<T>`	按照给定比较器选出的最小元素的 Optional， 或如果流为空则为 Optional.empty()	`Optional<Project> fattest = projectStream.collect(minBy(comparingInt(Project::getS`
`reducing`	归约操作产生的类型	从一个作为累加器的初始值开始，利用 BinaryOperator 与流中的元素逐个结合，从而将流归约为单个值	`int totalStars = projectStream.collect(reducing(0, Project::getStars, Integer::sum));`
`collectingAndThen`	转换函数返回的类型	包含另一个收集器，对其结果应用转换函数	`int howManyProjects = projectStream.collect(collectingAndThen(toList(), List::size));`
`groupingBy`	`Map<K, List<T>>`	根据项目的一个属性的值对流中的项目作问组，并将属性值作 为结果 Map 的键	`Map<String,List<Project>> projectByLanguage = projectStream.collect(groupingBy(Project::getLanguage)`
`partitioningBy`	`Map<Boolean,List<T>>`	根据对流中每个项目应用断言的结果来对项目进行分区	`Map<Boolean,List<Project>> vegetarianDishes = projectStream.collect(partitioningBy(Project::isVegeta`

转换类型

有一些收集器可以生成其他集合。比如前面已经见过的 `toList`，生成了 `java.util.List` 类的实例。 还有 `toSet` 和 `toCollection`，分别生成 `Set` 和 `Collection` 类的实例。 到目前为止， 我已经讲了很多流上的链式操作，但总有一些时候，需要最终生成一个集合——比如：

- 已有代码是为集合编写的，因此需要将流转换成集合传入；
- 在集合上进行一系列链式操作后，最终希望生成一个值；
- 写单元测试时，需要对某个具体的集合做断言。

使用 `Collectors.toCollection`，用定制的集合收集元素

```
stream.collect(Collectors.toCollection(TreeSet::new));
```

还可以利用收集器让流生成一个值。 `maxBy` 和 `minBy` 允许用户按某种特定的顺序生成一个值。

数据分区

分区是分组的特殊情况：由一个断言（返回一个布尔值的函数）作为分类函数，它称分区函数。 分区函数返回一个布尔值，这意味着得到的分组 `Map` 的键类型是 `Boolean`，于是它最多可以分为两组: true是一组，false是一组。

分区的好处在于保留了分区函数返回true或false的两套流元素列表。

并行流

并行流就是一个把内容分成多个数据块，并用不不同的线程分别处理每个数据块的流。最后合并每个数据块的计算结果。

将一个顺序执行的流转变成一个并发的流只要调用 `parallel()` 方法

```
public static long parallelSum(long n){
    return Stream.iterate(1L, i -> i
+1).limit(n).parallel().reduce(0L,Long::sum);
}
```

将一个并发流转成顺序的流只要调用 `sequential()` 方法

```
stream.parallel().filter(...).sequential().map(...).parallel().reduce();
```

这两个方法可以多次调用，只有最后一个调用决定这个流是顺序的还是并发的。

并发流使用的默认线程数等于你机器的处理器核心数。

通过这个方法可以修改这个值，这是全局属性。

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism",
"12");
```

并非使用多线程并行流处理数据的性能一定高于单线程顺序流的性能，因为性能受到多种因素的影响。 如何高效使用并发流的一些建议：

1. 如果不确定， 就自己测试。
2. 尽量使用基本类型的流 `IntStream`, `LongStream`, `DoubleStream`
3. 有些操作使用并发流的性能会比顺序流的性能更差，比如limit，findFirst，依赖元素顺序的操作在并发流中是极其消耗性能的。findAny的性能就会好很多，应为不依赖顺序。
4. 考虑流中计算的性能(Q)和操作的性能(N)的对比, Q表示单个处理所需的时间，N表示需要处理的数量，如果Q的值越大, 使用并发流的性能就会越高。
5. 数据量不大时使用并发流，性能得不到提升。
6. 考虑数据结构：并发流需要对数据进行分解，不同的数据结构被分解的性能时不一样的。

流的数据源和可分解性

源	可分解性
`ArrayList`	非常好
`LinkedList`	差
`IntStream.range`	非常好
`Stream.iterate`	差
`HashSet`	好
`TreeSet`	好

流的特性以及中间操作对流的修改都会对数据对分解性能造成影响。 比如固定大小的流在任务分解的时候就可以平均分配，但是如果有filter操作，那么流就不能预先知道在这个操作后还会剩余多少元素。

考虑终端操作的性能：如果终端操作在合并并发流的计算结果时的性能消耗太大，那么使用并发流提升的性能就会得不偿失。

Optional 类（★）

Optional 类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

Optional 是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional 类的引入很好的解决空指针异常。

Optional类的方法

方法	描述
`empty`	返回一个空的 Optional 实例
`filter`	如果值存在并且满足提供的断言， 就返回包含该值的 Optional 对象；否则返回一个空的 Optional 对象
`map`	如果值存在，就对该值执行提供的 mapping 函数调用
`flatMap`	如果值存在，就对该值执行提供的 mapping 函数调用，返回一个 Optional 类型的值，否则就返 回一个空的 Optional 对象
`get`	如果该值存在，将该值用 Optional 封装返回，否则抛出一个 NoSuchElementException 异常
`ifPresent`	如果值存在，就执行使用该值的方法调用，否则什么也不做
`isPresent`	如果值存在就返回 true，否则返回 false
`of`	将指定值用 Optional 封装之后返回，如果该值为 null，则抛出一个 NullPointerException 异常
`ofNullable`	将指定值用 Optional 封装之后返回，如果该值为 null，则返回一个空的 Optional 对象
`orElse`	如果有值则将其返回，否则返回一个默认值
`orElseGet`	如果有值则将其返回，否则返回一个由指定的 Supplier 接口生成的值
`orElseThrow`	如果有值则将其返回，否则抛出一个由指定的 Supplier 接口生成的异常

Nashorn, JavaScript 引擎

Nashorn 一个 javascript 引擎。

从JDK 1.8开始，Nashorn取代Rhino(JDK 1.6, JDK1.7)成为Java的嵌入式JavaScript引擎。Nashorn完全支持ECMAScript 5.1规范以及一些扩展。它使用基于JSR 292的新语言特性，其中包含在JDK 7中引入的invokedynamic，将JavaScript编译成Java字节码。

与先前的Rhino实现相比，这带来了2到10倍的性能提升。

新的日期时间 API（★★）

ZoneId

Java 8中的时区操作被很大程度上简化了，新的时区类 `java.time.ZoneId` 是原有的 `java.util.TimeZone` 类的替代品。 ZoneId对象可以通过 `ZoneId.of()` 方法创建，也可以通过 `ZoneId.systemDefault()` 获取系统默认时区：

```
ZoneId shanghaiZoneId = ZoneId.of("Asia/Shanghai");
ZoneId systemZoneId = ZoneId.systemDefault();
```

`of()` 方法接收一个“区域/城市”的字符串作为参数，你可以通过 `getAvailableZoneIds()` 方法获取所有合法的“区域/城市”字符串：

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
```

对于老的时区类 `TimeZone`，Java 8也提供了转化方法：

```
ZoneId oldToNewZoneId = TimeZone.getDefault().toZoneId();
```

有了 `ZoneId`，我们就可以将一个 `LocalDate`、`LocalTime` 或 `LocalDateTime` 对象转化为 `ZonedDateTime` 对象：

```
LocalDateTime localDateTime = LocalDateTime.now();
ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, shanghaiZoneId);
```

`ZonedDateTime` 对象由两部分构成，`LocalDateTime` 和 `ZoneId`，其中 `2018-03-03T15:26:56.147` 部分为 `LocalDateTime`，`+08:00[Asia/Shanghai]` 部分为ZoneId。

另一种表示时区的方式是使用 `ZoneOffset`，它是以当前时间和 **世界标准时间（UTC）/格林威治时间（GMT）** 的偏差来计算，例如：

```
ZoneOffset zoneOffset = ZoneOffset.of("+09:00");
LocalDateTime localDateTime = LocalDateTime.now();
OffsetDateTime offsetDateTime = OffsetDateTime.of(localDateTime, zoneOffset);
```

Instant

Instant类在Java日期与时间功能中，表示了时间线上一个确切的点，定义为距离初始时间的时间差（初始时间为GMT 1970年1月1日00:00）经测量一天有86400秒，从初始时间开始不断向前移动。

创建一个Instant实例

你可以通过Instant类的工厂方法创建一个Instant实例，例如你可以调用instant.now()来创建一个确切的表达当前时间的Instant对象：

```
Instant now = Instant.now();
```

另外也有一些其它方法能创建Instant，具体请查阅Java官方文档。

访问Instant的时间

一个Instant对象里有两个域：距离初始时间的秒钟数、在当前一秒内的第几纳秒，他们的组合表达了当前时间点。你可以通过以下两个方法得到它们的值：

```
long seconds =  now.getEpochSecond()
int nanos   =   now.getNano()
```

Instant的计算

Instant类有一些方法，可以用于获得另一Instant的值，例如：

- `plusSeconds()`
- `plusMillis()`
- `plusNanos()`
- `minusSeconds()`
- `minusMillis()`
- `minusNanos()`

我下面将向你展示两个例子，来说明这些方法如何使用：

```
Instant now     = Instant.now();
Instant later   = now.plusSeconds(3);
Instant earlier = now.minusSeconds(3);
```

第一行获得了一个Instant对象，表示当前时间。第二行创建了一个Instant表示三秒后，第三行创建了一个Instant表示三秒前。

> seconds 表示从 `1970-01-01 00:00:00` 开始到现在的秒数，nanos 表示纳秒部分（nanos的值不会超过999,999,999）

Clock

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 `System.currentTimeMillis()` 来获取当前的微秒数。 某一个特定的时间点也可以使用Instant类来表示，Instant 类也可以用来创建老的 `java.util.Date` 对象。

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

LocalDate

LocalDate类是Java 8中日期时间功能里表示一个本地日期的类，它的日期是无时区属性的。 可以用来表示生日、节假日期等等。这个类用于表示一个确切的日期，而不是这个日期所在的时间（如java.util.Date中的2000.01.01表示的实际是这一天的00:00这个瞬间）。

LocalDate类位于java.time包下，类名叫java.time.LocalDate，创建出来的实例也是不可变对象，所以涉及它的计算方法将返回一个新的LocalDate。

创建一个LocalDate实例

我们有多种方式可以创建出 `LocalDate` 实例。第一种方法是使用 `now()` 方法获得值为今天当日的 `LocalDate` 对象：

```
LocalDate localDate = LocalDate.now();
```

另一种方法是使用年月日信息构造出LocalDate对象：

```
LocalDate localDate2 = LocalDate.of(2018, 7, 19);
```

LocalDate 的 `of()` 方法创建出一个指定年月日的日期，并且没有时区信息。

访问日期信息

可以用如下方法访问LocalDate中的日期信息：

```
int   year       = localDate.getYear();
Month month      = localDate.getMonth();
int   dayOfMonth = localDate.getDayOfMonth();
int   dayOfYear  = localDate.getDayOfYear();
DayOfWeek dayOfWeek = localDate.getDayOfWeek();
```

可以注意到getMonth()与getDayOfWeek()方法返回了一个枚举类型代替一个int。你可以通过枚举类型中的getValue()来获得信息。

LocalDate计算

你可以进行一堆简单的日期计算，只要使用如下的方法：

- `plusDays()`
- `plusWeeks()`
- `plusMonths()`
- `plusYears()`
- `minusDays()`
- `minusWeeks()`
- `minusMonths()`
- `minusYears()`

以下举几个使用的例子来帮助理解使用：

```
LocalDate d  = LocalDate.of(2018, 7, 19);
LocalDate d1 = localDate.plusYears(3);
LocalDate d2 = localDate.minusYears(3);
```

1. 第一行创建出一个新的LocalDate对象d，表示2018.7.19。
2. 第二行创建了值等于d日期3年后的LocalDate对象，第三行也是一样，只是值改为d日期的三年前。

LocalTime

LocalTime类是Java 8中日期时间功能里表示一整天中某个时间点的类，它的时间是无时区属性的（早上10点等等）。比如你需要描述学校几点开学，这个时间不涉及在什么城市，这个描述是对任何国家城市都适用的，此时使用无时区的LocalTime就足够了。 LocalTime类的对象也是不可变的，所以计算方法会返回一个新的LocalTime实例。

创建一个LocatTime实例

有多种方式可以新建LocalTime实例。比如使用当前时间作为值新建对象：

```
1 LocalTime localTime = LocalTime.now();
```

另一种方式是使用指定的时分秒和纳秒来新建对象：

```
1 LocalTime localTime2 = LocalTime.of(21, 30, 59, 11001);
```

也有另一种版本的 `of()` 方法只需要小时分钟两项，或时分秒三项值作为参数。

访问LocalTime对象的时间

你可以通过这些方法访问其时、分、秒、纳秒：

- `getHour()`
- `getMinute()`
- `getSecond()`
- `getNano()`

LocalTime的计算

LocalTime类包含一系列方法，能帮你完成时间计算：

- `plusHours()`
- `plusMinutes()`
- `plusSeconds()`
- `plusNanos()`
- `minusHours()`
- `minusMinutes()`
- `minusSeconds()`
- `minusNanos()`

以下举一个例子：

```
LocalTime localTime2 = LocalTime.of(21, 30, 59, 11001);
LocalTime localTimeLater   = localTime.plusHours(3);
LocalTime localTimeEarlier = localTime.minusHours(3);
```

1. 第一行新建一个LocalTime实例，表示21:30:50的第11001纳秒。
2. 第二行新建了一个LocalTime实例表示这个时间的三小时后，第三行表示三小时前。
3. LocalTime类是Java 8中日期时间功能里表示一整天中某个时间点的类，它的时间是无时区属性的（早上10点等等）。比如你需要描述学校几点开学，这个时间不涉及在什么城市，这个描述是对任何国家城市都适用的，此时使用无时区的LocalTime就足够了。

LocalTime类的对象也是不可变的，所以计算方法会返回一个新的LocalTime实例。

LocalDateTime

LocalDateTime类是Java 8中日期时间功能里，用于表示当地的日期与时间的类，它的值是无时区属性的。你可以将其视为Java 8中LocalDate与LocalTime两个类的结合。

LocalDateTime类的值是不可变的，所以其计算方法会返回一个新的LocalDateTime实例。

创建一个LocatDateTime实例

可以通过LocalDateTime的静态工厂方法来创建LocalDateTime实例。以下举例使用 `now()` 方法创建：

```
LocalDateTime localDateTime = LocalDateTime.now();
```

另一种方式是使用指定的年月日、时分秒、纳秒来新建对象：

```
LocalDateTime localDateTime2 = LocalDateTime.of(2018, 7, 19, 13, 55, 36, 123);
```

访问LocalDateTime对象的时间

你可以通过这些方法访问其日期时间：

- `getYear()`
- `getMonth()`
- `getDayOfMonth()`
- `getDayOfWeek()`
- `getDayOfYear()`
- `getHour()`
- `getMinute()`
- `getSecond()`
- `getNano()`

这些方法中有一些返回int有一些返回枚举类型，你可以通过枚举类型中的 `getValue()` 方法来获得int值。

LocalDateTime的计算

LocalDateTime 类包含一系列方法，能帮你完成时间计算：

- `plusYears()`
- `plusMonths()`
- `plusDays()`
- `plusHours()`
- `plusMinutes()`
- `plusSeconds()`
- `plusNanos()`
- `minusYears()`
- `minusMonths()`
- `minusDays()`
- `minusHours()`
- `minusMinutes()`
- `minusSeconds()`
- `minusNanos()`

以下举一个例子：

```
LocalDateTime localDateTime  = LocalDateTime.now();
LocalDateTime localDateTime1 = localDateTime.plusYears(3);
LocalDateTime localDateTime2 = localDateTime.minusYears(3);
```

1. 第一行新建一个LocalDateTime实例表示当前这个时间。
2. 第二行新建了一个LocalDateTime实例表示三年后。
3. 第三行也新建了一个LocalDateTime实例表示三小时前。

ZonedDateTime

ZonedDateTime类是Java 8中日期时间功能里，用于表示带时区的日期与时间信息的类。可以用于表示一个真实事件的开始时间，如某火箭升空时间等等。

ZonedDateTime 类的值是不可变的，所以其计算方法会返回一个新的ZonedDateTime 实例。

创建一个ZonedDateTime实例

有多种方式可以新建ZonedDateTime实例。比如使用当前时间作为值新建对象：

```
ZonedDateTime dateTime = ZonedDateTime.now();
```

另一种方式是使用指定的年月日、时分秒、纳秒以及时区ID来新建对象：

```
ZoneId zoneId = ZoneId.of("UTC+1");
ZonedDateTime dateTime2 = ZonedDateTime.of(2018, 7, 19, 11, 45, 59, 1234,
zoneId);
```

访问ZonedDateTime对象的时间

你可以通过这些方法访问其日期时间：

- `getYear()`
- `getMonth()`
- `getDayOfMonth()`
- `getDayOfWeek()`
- `getDayOfYear()`
- `getHour()`
- `getMinute()`
- `getSecond()`
- `getNano()`

这些方法中有一些返回int有一些返回枚举类型，但可以通过枚举类型中的getValue()方法来获得int值。

ZonedDateTime的计算

ZonedDateTime类包含一系列方法，能帮你完成时间计算：

- `plusYears()`
- `plusMonths()`
- `plusDays()`
- `plusHours()`
- `plusMinutes()`
- `plusSeconds()`
- `plusNanos()`
- `minusYears()`
- `minusMonths()`
- `minusDays()`
- `minusHours()`
- `minusMinutes()`
- `minusSeconds()`
- `minusNanos()`

但注意计算时，若不巧跨越了夏令时（会补一小时或减一小时），可能得不到希望的结果。一个替代的正确做法是使用Period：

```
1  ZonedDateTime zoneDateTime = previousDateTime.plus(Period.ofDays(3));
```

时区

时区是用ZoneId类表示的，你可以使用ZoneId.now()或ZoneId.of(“xxx”)来实例化：

```
1  ZoneId zoneId = ZoneId.of("UTC+1");
```

传给 `of()` 方法的参数是时区的ID，如“UTC+1”指距离UTC（格林威治时间）有一小时的时差，你可以使用你想要的时差来表示ZoneId（如+1与-5等等） 你也可以使用另一种方式表示zone id，即使用地区名字，也是可以的：

```
ZoneId zoneId2 = ZoneId.of("Europe/Copenhagen");
ZoneId zoneId3 = ZoneId.of("Europe/Paris");
```

DateTimeFormatter

DateTimeFormatter类是Java 8中日期时间功能里，用于解析和格式化日期时间的类，位于 `java.time.format` 包下。

预定义的DateTimeFormatter实例

DateTimeFormatter类包含一系列预定义（常量）的实例，可以解析和格式化一些标准时间格式。这将让你免除麻烦的时间格式定义，类中包含如下预定义的实例：

```
BASIC_ISO_DATE

ISO_LOCAL_DATE
ISO_LOCAL_TIME
ISO_LOCAL_DATE_TIME

ISO_OFFSET_DATE
ISO_OFFSET_TIME
ISO_OFFSET_DATE_TIME

ISO_ZONED_DATE_TIME

ISO_INSTANT

ISO_DATE
ISO_TIME
ISO_DATE_TIME

ISO_ORDINAL_TIME
ISO_WEEK_DATE

RFC_1123_DATE_TIME
```

每个预定义的DateTimeFormatter实例都有不同的日期格式，具体的可以查阅Java官方文档。

> PS：推荐使用下期分享的 `Hutool` 工具类做这些格式化工作，格式更加丰富。

格式化日期

当你获取一个DateTimeFormatter实例后，就可以用format()方便来将一个日期格式化为某种字符串，例如：

```
DateTimeFormatter formatter = DateTimeFormatter.BASIC_ISO_DATE;
String formattedDate = formatter.format(LocalDate.now());
System.out.println(formattedDate);
```

这个样例把LocalDate对象格式化了，并输出20180719，这个输出表示现在2018年，7月19日。 再举一个关于ZonedDateTime的例子：

```
DateTimeFormatter formatter = DateTimeFormatter.BASIC_ISO_DATE;
String formattedZonedDate = formatter.format(ZonedDateTime.now());
System.out.println("formattedZonedDate = " + formattedZonedDate);
```

这个例子会输出：20180719+0800 表示今年2018年，7月19日，位于UTC+8时区。

Duration

一个Duration对象表示两个Instant间的一段时间，是在Java 8中加入的新功能。

一个Duration实例是不可变的，当创建出对象后就不能改变它的值了。你只能通过Duration的计算方法，来创建出一个新的Durtaion对象。你会在之后的教程中见到的。

创建Duration实例

使用 `Duration` 类的工厂方法来创建一个 `Duration` 对象，以下是一个使用 `between()` 的例子：

```
Instant first = Instant.now();
// 耗时操作，或者 Sleep 一段时间
Instant second = Instant.now();
Duration duration = Duration.between(first, second);
```

访问Duration的时间

一个Duration对象里有两个域：纳秒值（小于一秒的部分），秒钟值（一共有几秒），他们的组合表达了时间长度。注意屯使用System.getCurrentTimeMillis()时不同，Duration不包含毫秒这个属性。你可以通过以下两个方法得到它们的值：

```
long seconds =  duration.getSeconds()
int nanos    =  duration.getNano()
```

你也可以转换整个时间到其它单位如纳秒、分钟、小时、天：

- `toNanos()`
- `toMillis()`
- `toMinutes()`
- `toHours()`
- `toDays()`

举例而言：`toNanos()` 与 `getNano()` 不同，`toNanos()` 获得的是 `Duration` 整个时间共有多少纳秒， 而 `getNano()` 只是获得这段时间中小于一秒的部分。

Duration计算

Duration类包含一系列的计算方法：

- `plusNanos()`
- `plusMillis()`
- `plusSeconds()`
- `plusMinutes()`
- `plusHours()`
- `plusDays()`
- `minusNanos()`
- `minusMillis()`
- `minusSeconds()`
- `minusMinutes()`
- `minusHours()`
- `minusDays()`

这些方法所做的事都是相似的，这里展示一个加减的例子：

```
Duration start = ... //obtain a start duration
Duration added      = start.plusDays(3);
Duration subtracted = start.minusDays(3);
```

1. 第一行创建了一个Duration对象叫start，具体怎么创建可以参考前面的代码。
2. 第二三行样例创建了两个新的Duration，通过调用start的加减操作，使得added对象表示的时间比start多三天，而substracted则少三天。

所有的计算方法都会返回一个新的Duration，以保证Duration的不可变属性。

```
long days = duration.toDays();              // 这段时间的总天数
long hours = duration.toHours();            // 这段时间的小时数
long minutes = duration.toMinutes();        // 这段时间的分钟数
long seconds = duration.getSeconds();       // 这段时间的秒数
long milliSeconds = duration.toMillis();    // 这段时间的毫秒数
long nanoSeconds = duration.toNanos();      // 这段时间的纳秒数
```

其他操作

增加和减少日期

Java 8中的日期/时间类都是不可变的，这是为了保证线程安全。当然，新的日期/时间类也提供了方法用于创建对象的可变版本，比如增加一天或者减少一天：

```
LocalDate date = LocalDate.of(2017, 1, 5);          // 2017-01-05

LocalDate date1 = date.withYear(2016);              // 修改为 2016-01-05
LocalDate date2 = date.withMonth(2);                // 修改为 2017-02-05
LocalDate date3 = date.withDayOfMonth(1);           // 修改为 2017-01-01

LocalDate date4 = date.plusYears(1);                // 增加一年 2018-01-05
LocalDate date5 = date.minusMonths(2);              // 减少两个月 2016-11-05
LocalDate date6 = date.plus(5, ChronoUnit.DAYS);    // 增加5天 2017-01-10
```

上面例子中对于日期的操作比较简单，但是有些时候我们要面临更复杂的时间操作，比如将时间调到下一个工作日， 或者是下个月的最后一天，这时候我们可以使用 `with()` 方法的另一个重载方法，它接收一个TemporalAdjuster参数， 可以使我们更加灵活的调整日期：

```
LocalDate date7 = date.with(nextOrSame(DayOfWeek.SUNDAY));      // 返回下一个距离当前时间最近的星期日
LocalDate date9 = date.with(lastInMonth(DayOfWeek.SATURDAY));   // 返回本月最后一个星期六
```

要使上面的代码正确编译，你需要使用静态导入 `TemporalAdjusters` 对象：

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

`TemporalAdjusters` 类中包含了很多静态方法可以直接使用，下面的表格列出了一些方法：

方法名	描述
`dayOfWeekInMonth`	返回同一个月中每周的第几天
`firstDayOfMonth`	返回当月的第一天
`firstDayOfNextMonth`	返回下月的第一天
`firstDayOfNextYear`	返回下一年的第一天
`firstDayOfYear`	返回本年的第一天
`firstInMonth`	返回同一个月中第一个星期几
`lastDayOfMonth`	返回当月的最后一天
`lastDayOfNextMonth`	返回下月的最后一天
`lastDayOfNextYear`	返回下一年的最后一天
`lastDayOfYear`	返回本年的最后一天
`lastInMonth`	返回同一个月中最后一个星期几
`next / previous`	返回后一个/前一个给定的星期几
`nextOrSame / previousOrSame`	返回后一个/前一个给定的星期几，如果这个值满足条件，直接返回

如果上面表格中列出的方法不能满足你的需求，你还可以创建自定义的 `TemporalAdjuster` 接口的实现， `TemporalAdjuster` 也是一个函数式接口，所以我们可以使用Lambda表达式：

```
@FunctionalInterface
public interface TemporalAdjuster {
    Temporal adjustInto(Temporal temporal);
}
```

比如给定一个日期，计算该日期的下一个工作日（不包括星期六和星期天）：

```
LocalDate date = LocalDate.of(2018, 7, 19);
date.with(temporal -> {
    // 当前日期
    DayOfWeek dayOfWeek = DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));

    // 正常情况下，每次增加一天
    int dayToAdd = 1;

    // 如果是星期五，增加三天
    if (dayOfWeek == DayOfWeek.FRIDAY) {
```

```
        dayToAdd = 3;
    }

    // 如果是星期六，增加两天
    if (dayOfWeek == DayOfWeek.SATURDAY) {
        dayToAdd = 2;
    }

    return temporal.plus(dayToAdd, ChronoUnit.DAYS);
});
```

其他历法

Java中使用的历法是ISO 8601日历系统，它是世界民用历法，也就是我们所说的公历。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。

此外Java 8还提供了4套其他历法（很奇怪为什么没有汉族人使用的农历），每套历法都包含一个日期类，分别是：

- `ThaiBuddhistDate`：泰国佛教历
- `MinguoDate`：中华民国历
- `JapaneseDate`：日本历
- `HijrahDate`：伊斯兰历

每个日期类都继承`ChronoLocalDate`类，所以可以在不知道具体历法的情况下也可以操作。不过这些历法一般不常用，除非是有某些特殊需求情况下才会使用。

这些不同的历法也可以用于向公历转换：

```
LocalDate date = LocalDate.now();
JapaneseDate jpDate = JapaneseDate.from(date);
```

由于它们都继承ChronoLocalDate类，所以在不知道具体历法情况下，可以通过ChronoLocalDate类操作日期：

```
Chronology jpChronology = Chronology.ofLocale(Locale.JAPANESE);
ChronoLocalDate jpChronoLocalDate = jpChronology.dateNow();
```

我们在开发过程中应该尽量避免使用`ChronoLocalDate`，尽量用与历法无关的方式操作时间，因为不同的历法计算日期的方式不一样，比如开发者会在程序中做一些假设，假设一年中有12个月，如果是中国农历中包含了闰月，一年有可能是13个月，但开发者认为是12个月，多出来的一个月属于明年的。

再比如假设年份是累加的，过了一年就在原来的年份上加一，但日本天皇在换代之后需要重新纪年，所以过了一年年份可能会从1开始计算。

在实际开发过程中建议使用 `LocalDate`，包括存储、操作、业务规则的解读；除非需要将程序的输入或者输出本地化，这时可以使用 `ChronoLocalDate` 类。

Base64

在Java 8中，Base64编码已经成为Java类库的标准。

Java 8 内置了 Base64 编码的编码器和解码器，这样不需要使用第三方库就可以进行Base64编码。

Base64工具类提供了一套静态方法获取下面三种BASE64编解码器：

- **基本**：输出被映射到一组字符A-Za-z0-9+/，编码不添加任何行标，输出的解码仅支持A-Za-z0-9+/。
- **URL**：输出映射到一组字符A-Za-z0-9+_，输出是URL和文件。
- **MIME**：输出隐射到MIME友好格式。输出每行不超过76字符，并且使用'\r'并跟随'\n'作为分割。编码输出最后没有行分割。

示例：

```
final String text = "测试Base64编码";

final String encoded = Base64
        .getEncoder()
        .encodeToString(text.getBytes(StandardCharsets.UTF_8));
System.out.println(encoded);

final String decoded = new String(
        Base64.getDecoder().decode(encoded),
        StandardCharsets.UTF_8);
System.out.println(decoded);
```

参考资料

- Java 8 新特性
- SimpleDateFormat的线程安全问题与解决方案
- 为什么SimpleDateFormat不是线程安全的?
- Java获取N天前，N天后的日期（如3天）
- What's New in JDK 8