**Java8**

Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。

Stream有以下特性及优点：

--无存储。Stream不是一种数据结构，它只是某种数据源的一个视图，数据源可以是一个数组，Java容器或I/O channel等。

--为函数式编程而生。对Stream的任何修改都不会修改背后的数据源，比如对Stream执行过滤操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新Stream。

--惰式执行。Stream上的操作并不会立即执行，只有等到用户真正需要结果（执行最终操作才会回溯中间操作）的时候才会执行。

--可消费性。Stream只能被“消费”一次，一旦遍历过就会失效，就像容器的迭代器那样，想要再次遍历必须重新生成。

--方便的并行处理。并行流的内部使用了默认的 ForkJoinPool 分支/合并框架，它的默认线程数量就是你的处理器数量，这个值是由 Runtime.getRuntime().availableProcessors() 得到的（当然我们也可以全局设置这个值）。

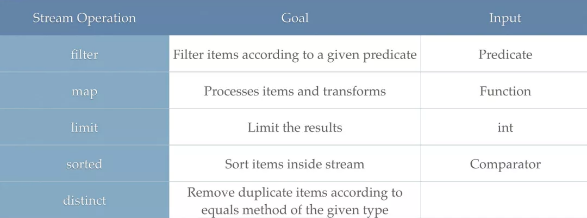
对于流的处理，主要有三种关键性操作：分别是流的创建、中间操作（intermediate operation）以及最终操作(terminal operation)。

流的创建

1、通过已有的集合来创建流（集合类的stream方法）

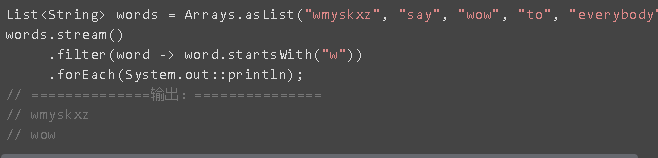
2、通过Stream创建流（通过Stream的of方法）

**中间操作**

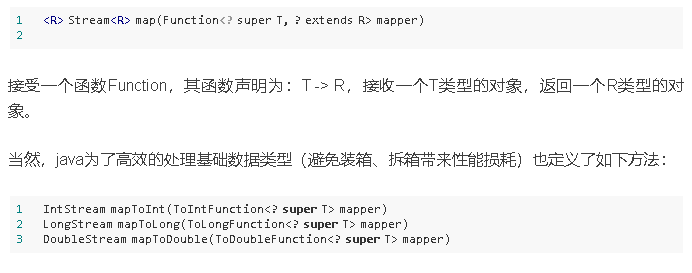


**filter**

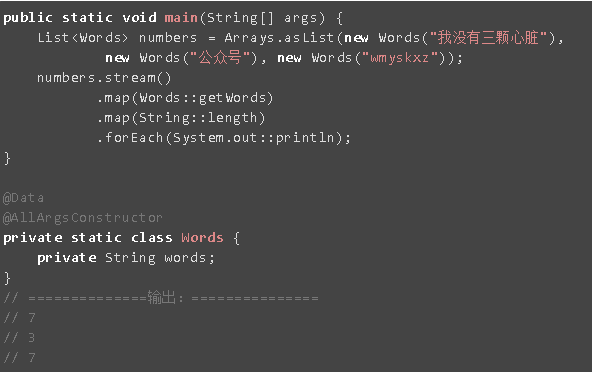
filter 方法用于通过设置的条件过滤出元素，该操作会接受一个返回 boolean 的函数作为参数，并返回一个包含所有符合该条件的流。



**Map**



map 方法用于映射每个元素到对应的结果，会接受一个函数作为参数。这个函数会被应用到每个元素身上，并将其映射成一个新的函数。



**limit/skip**

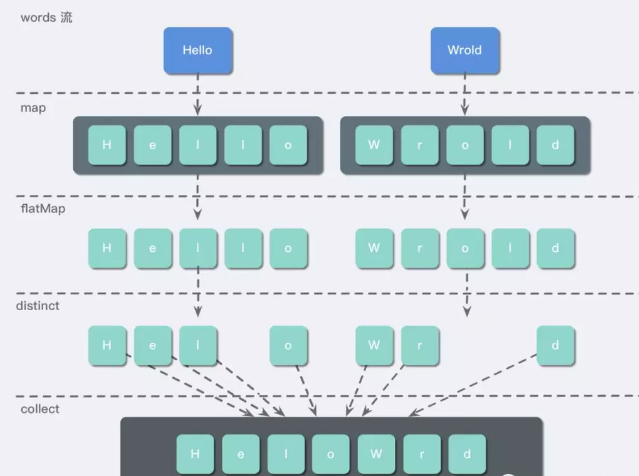
limit 返回 Stream 的前面 n 个元素；skip 则是扔掉前 n 个元素。

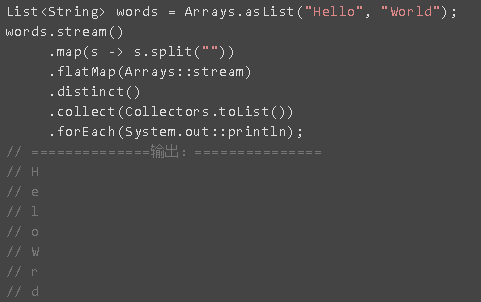
**sorted**

sorted 方法用于对流进行排序。

**flatMap：流的扁平化，将多个流合并成一个流**

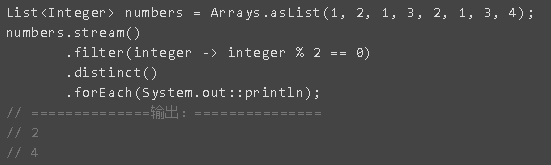
使用 flatMap 方法的效果是，各个数组并不是分别映射成一个流，而是映射成流的内容。一言蔽之就是 flatMap 让你一个流中的每个值都转换成另一个流，然后把所有的流连接起来成为一个流；





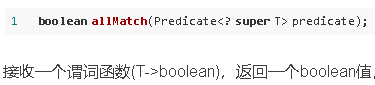
**distinct**

distinct主要用来去重，返回一个元素各异（根据流所生成的元素的 hashCode 和 equals 方法实现）的流



**查找和匹配**

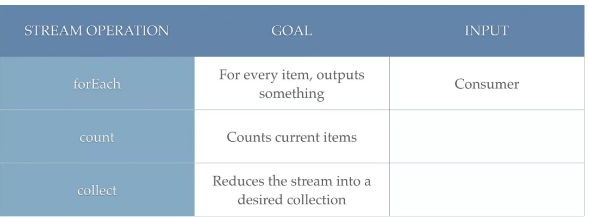
allMatch、anyMatch、noneMatch、findFirst 和 findAny



**最终操作**

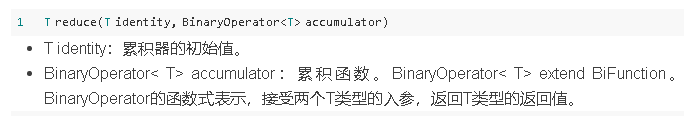
最终操作会消耗流，产生一个最终结果。也就是说，在最终操作之后，不能再次使用流，也不能在使用任何中间操作，否则将抛出异常：

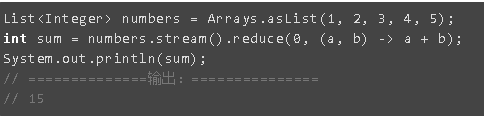


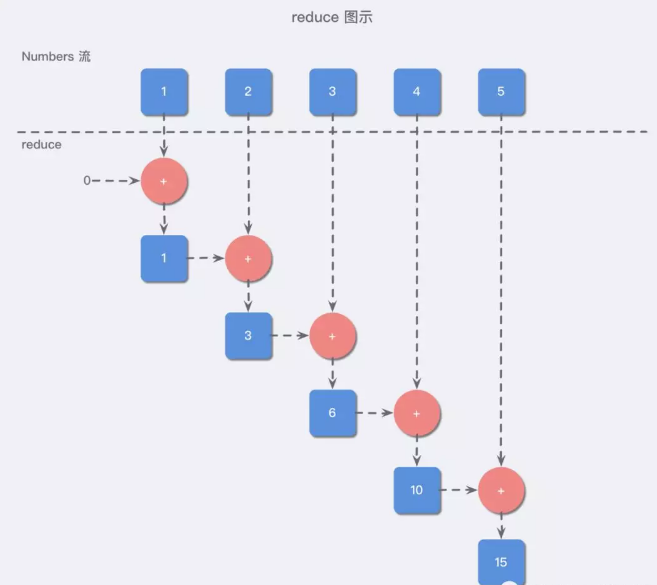


**归约：reduce**

**有初始值**





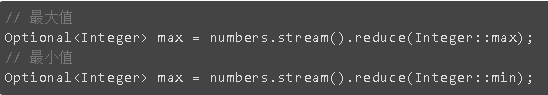


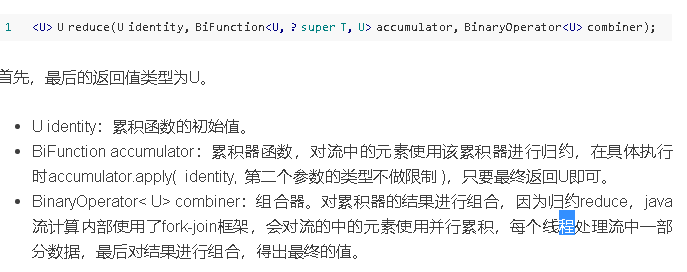
**无初始值**

它不接受初始值，但是会返回一个 Optional 对象(考虑到流中没有任何元素的情况)：



**最大值和最小值**





三个参数的reduce函数的使用场景：接下来还是用求和的例子来展示其使用场景。在java多线程编程模型中，引入了fork-join框架，就是对一个大的任务进行先拆解，用多线程分别并行执行，最终再两两进行合并，得出最终的结果。reduce函数的第三个函数，就是组合这个动作

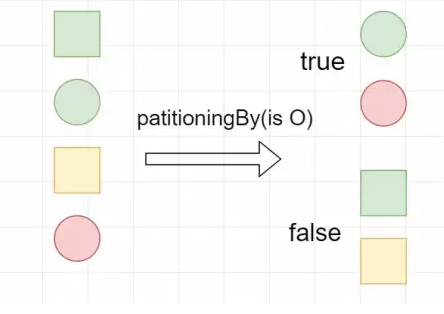


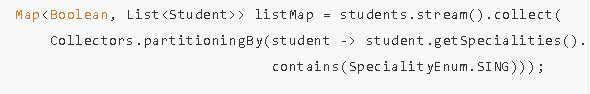
**collect**

**collect就是一个归约操作，可以接受各种做法作为参数，将流中的元素累积成一个汇总结果**

**转换成块**

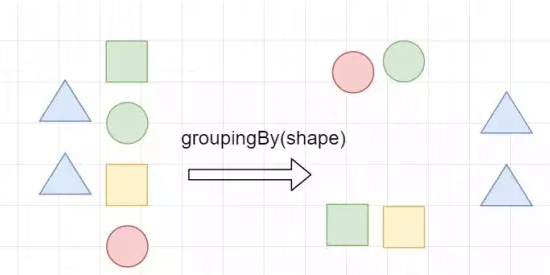
常用的流操作是将其分解成两个集合，Collectors.partitioningBy帮我们实现了，接收一个Predicate函数式接口

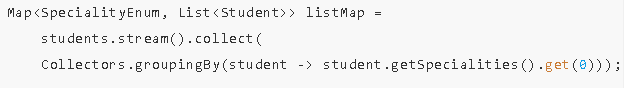
。



**数据分组**

数据分组是一种更自然的分割数据操作，与将数据分成 ture 和 false 两部分不同，可以使用任意值对数据分组。Collectors.groupingBy接收一个Function做转换。





### **字符串拼接**



joining接收三个参数，第一个是分界符，第二个是前缀符，第三个是结束符。也可以不传入参数Collectors.joining()，这样就是直接拼接。

**性能测试**

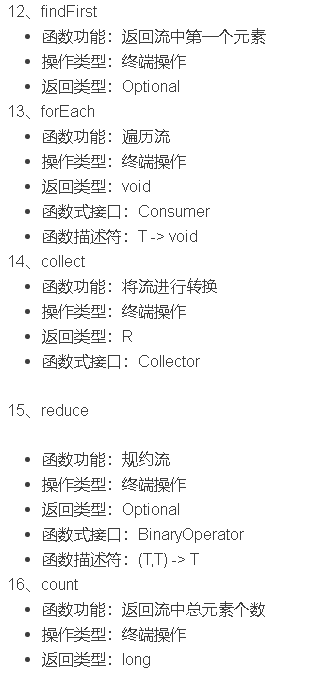
1、对于简单操作推荐使用外部迭代手动实现，

2、对于复杂操作（reduce归约操作），推荐使用Stream API，

3、在多核情况下，推荐使用并行Stream API来发挥多核优势

4、单核情况下不建议使用并行Stream API。







**Java 8 Time Api**

--线程安全 - Date 和Calendar类不是线程安全的，使开发者难以调试这些api的并发问题，需要编写额外的代码来处理线程安全。Java 8中引入的新的Date和Time API是不可变的和线程安全的，使得这些痛点得以解决。

--API设计和易于理解 - 旧的时间api非常难以理解，操作都非常复杂，非常绕口，没有提供一些常用的解析转换方法。新的时间API是以ISO为中心的，并遵循 date, time, duration 和 periods的一致域模型。提供了一些非常实用方法以支持最常见的操作。不再需要我们自己封装一些时间操作类。

--ZonedDate和Time - 在旧的时间api中开发人员必须编写额外的逻辑来处理旧API的时区逻辑，而使用新的API，可以使用 Local和ZonedDate / Time API来处理时区。无需过多关心时区转换问题。

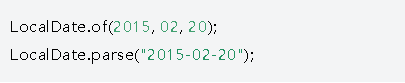
**LocalDate**

LocalDate表示在ISO格式（YYYY-MM-DD）下的不带具体时间的日期。

获取当前系统时钟下的日期



表示特定日，月和年的LocalDate可以使用“ of ”方法或使用“ parse ”方法获得。



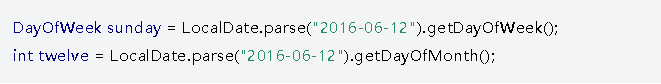
获取当前本地日期并添加一天



获取当前日期并减去一个月



获取星期几和月中的某天



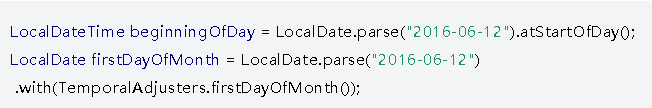
测试一个日期是否发生在闰年



判断日期的先后



代表给定日期的一天的开始（2016-06-12T00：00）和代表月初的LocalDate（2016-06-01）



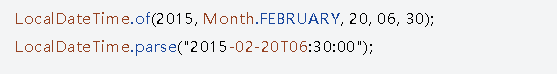
**LocalTime**（基本同上）

一天中的最大，最小和中午时间可以通过LocalTime类中的常量获得



**LocalDateTime**

使用工厂“of”和“parse”方法创建实例



**ZonedDateTime API**

获取下“亚洲/上海”时区



获取所有的时区：



LocalDateTime转化为特定的时区中的时间

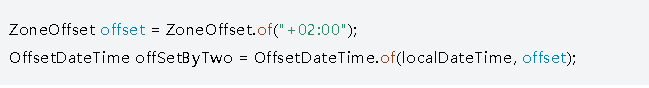


ZonedDateTime提供解析方法来获取时区的特定日期时间



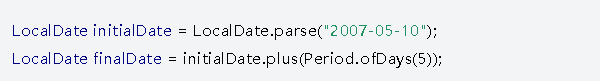
**OffsetDateTime**

OffsetDateTime是具有偏移量的日期时间的不可变表示形式。此类存储所有日期和时间字段，精确到纳秒，以及从UTC/格林威治的偏移量。可以使用ZoneOffset创建OffsetDateTime实例。



**Period**

用于修改给定的日期的值或者获取两个日期之间的差值

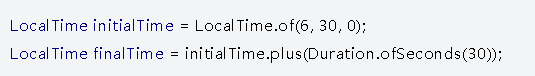






**Duration**

用来处理时间差







**与日期和日历的兼容性**

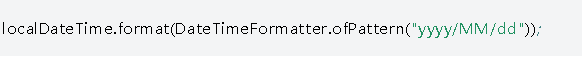
Java 8添加了toInstant（）方法，该方法有助于将旧API中的Date和Calendar实例转换为新的Date Time API





**日期和时间格式化**

传递ISO日期格式以格式化本地日期



格式样式传递为SHORT，LONG或MEDIUM作为格式化选项的一部分

