<http://www.cnblogs.com/CarpenterLee/category/965121.html>

**一、Java Stream API入门篇**

<http://www.importnew.com/24237.html>

**1、概述**

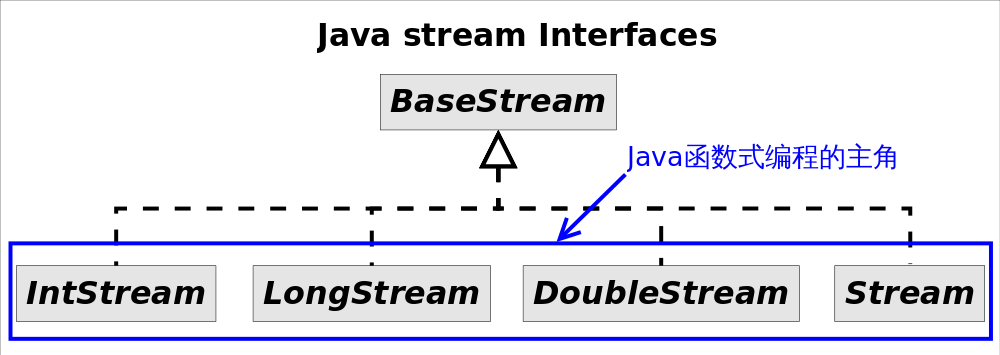
函数式编程：

* **代码简洁：**函数式编程写出的代码简洁且意图明确，使用stream接口让你从此告别for循环。
* **多核友好：**Java函数式编程使得编写并行程序从未如此简单，你需要的全部就是调用一下parallel()方法。

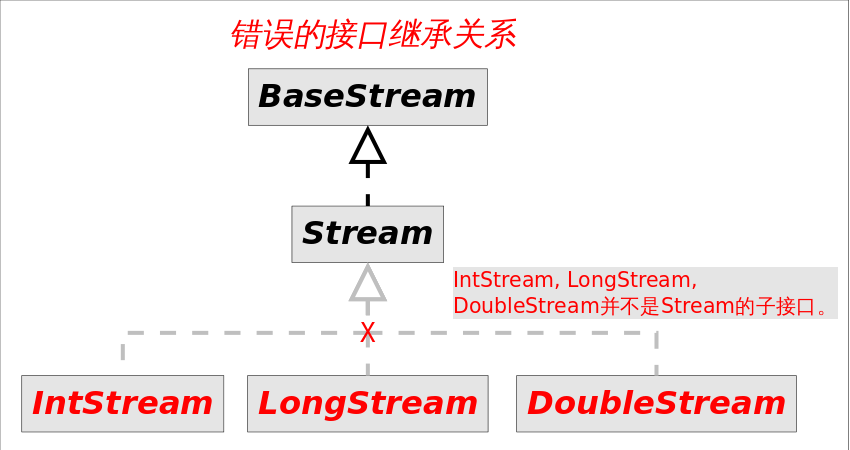
stream并不是**某种数据结构**，它只是**数据源的一种视图**。这里的数据源可以是一个**数组、Java容器或I/O channel**等。正因如此要得到一个stream通常不会手动创建，而是调用对应的工具方法，比如：

* 调用**Collection.stream()**或者**Collection.parallelStream()**方法；
* 调用**Arrays.stream(T[] array)**方法；

常见的**stream接口继承关系**如图：



图中4种stream接口继承自**BaseStream**，其中IntStream, LongStream, DoubleStream对应**三种基本类型**（int, long, double，注意不是包装类型），**Stream对应所有剩余类型的stream视图**。为不同数据类型设置不同stream接口，可以1.提高性能，2.增加特定接口函数。



你可能会奇怪**为什么不把IntStream等设计成Stream的子接口？**毕竟这接口中的方法名大部分是一样的。答案是这些方法的名字虽然相同，但是返回类型不同，如果设计成父子接口关系，这些方法将不能共存，因为**Java不允许只有返回类型不同的方法重载**。

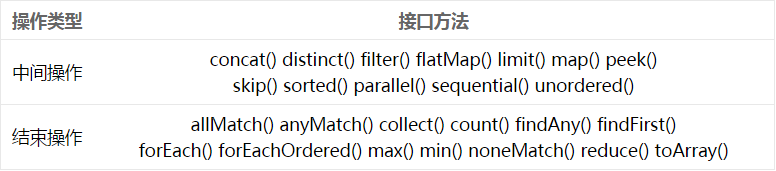
虽然大部分情况下stream是容器调用Collection.stream()方法得到的，但stream和collections有以下不同：

* **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。

对stream的操作分为为两类，**中间操作**(intermediate operations)和**结束操作**(terminal operations)，二者特点是：

* **中间操作总是会惰式执行**，调用中间操作只会生成一个标记了该操作的新stream，仅此而已。
* **结束操作会触发实际计算**，计算发生时会把所有中间操作积攒的操作以pipeline的方式执行，这样可以减少迭代次数。计算完成之后stream就会失效。

如果你熟悉**Apache Spark RDD**，对stream的这个特点应该不陌生。下表汇总了Stream接口的部分常见方法：



区分中间操作和结束操作最简单的方法，就是**看方法的返回值**，返回值为stream的大都是中间操作，否则是结束操作。

**stream方法使用：**

stream跟函数接口关系非常紧密，没有函数接口stream就无法工作。回顾一下：**函数接口是指内部只有一个抽象方法的接口**。通常**函数接口出现的地方都可以使用Lambda表达式**，所以不必记忆函数接口的名字。

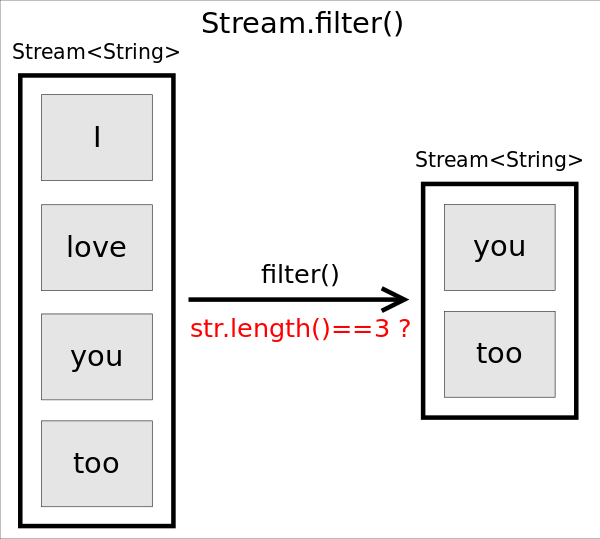
**2、forEach()**

我们对forEach()方法并不陌生，在Collection中我们已经见过。方法签名为void forEach(Consumer<? super E> action)，作用是**对容器中的每个元素执行action指定的动作，也就是对元素进行遍历**。

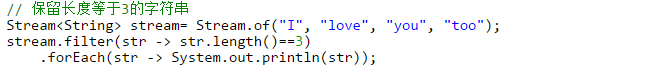


由于forEach()是结束方法，上述代码会立即执行，输出所有字符串。

**3、filter()**

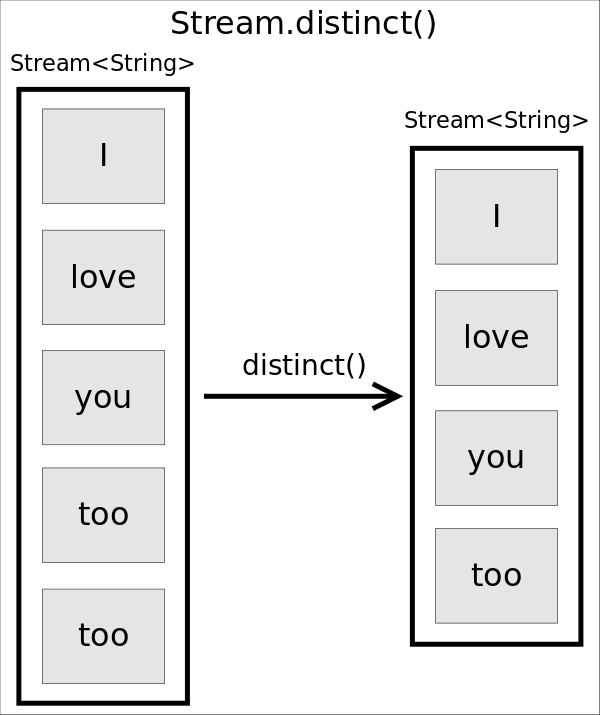


函数原型为**Stream<T> filter(Predicate<? super T> predicate)**，作用是**返回一个只包含满足predicate条件（过滤条件）元素的Stream**。



上述代码将输出为长度等于3的字符串you和too。注意，由于**filter()是个中间操作**（返回stream对象），如果只调用filter()不会有实际计算，因此也不会输出任何信息。

**4、distinct()**



函数原型为**Stream<T> distinct()**，作用是返回一个去除重复元素之后的Stream。



上述代码会输出去掉一个too之后的其余字符串。

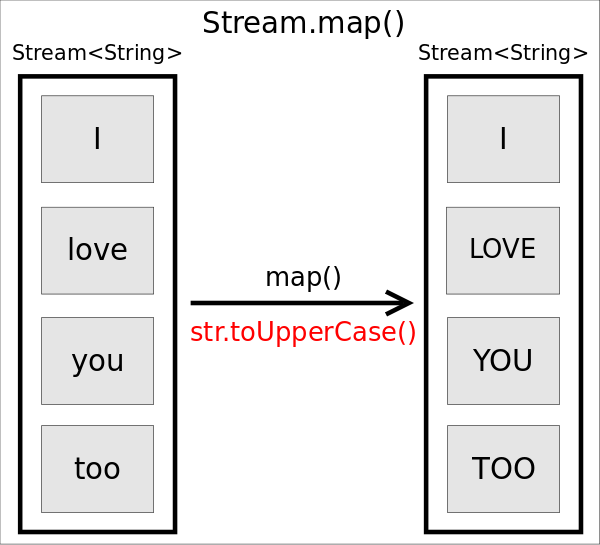
**5、sorted()：排序**

排序函数有两个，一个是用**自然顺序排序**，一个是使用**自定义比较器排序**，函数原型分别为**Stream<T>　sorted()**和**Stream<T>　sorted(Comparator<? super T> comparator)**。



上述代码将输出**按照长度升序排序后的字符串**，结果完全在预料之中。

**6、map()：对stream中的所有数据进行操作或转换**

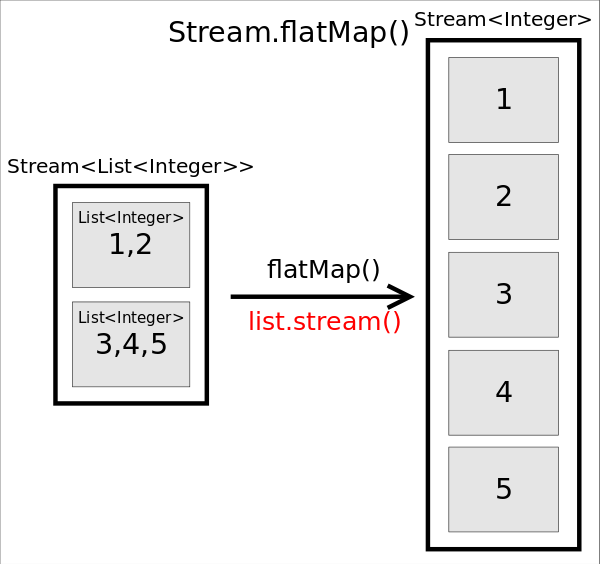


函数原型为**<R> Stream<R> map(Function<? super T,? extends R> mapper)**，作用是**返回一个对当前所有元素执行执行mapper之后的结果组成的Stream**。直观的说，就是对每个元素按照某种操作进行转换，转换前后Stream中元素的个数不会改变，但元素的类型取决于转换之后的类型。



上述代码将输出原字符串的大写形式。

**7、flatMap()：数据展开操作**



函数原型为**<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)**，作用是**对每个元素执行mapper指定的操作，并用所有mapper返回的Stream中的元素组成一个新的Stream作为最终返回结果**。说起来太拗口，通俗的讲flatMap()的作用就相当于把原stream中的所有元素都”摊平”之后组成的Stream，**转换前后元素的个数和类型都可能会改变**。



上述代码中，原来的stream中有两个元素，分别是两个List<Integer>，执行flatMap()之后，将每个List都“摊平”成了一个个的数字，所以会新产生一个由5个数字组成的Stream。所以最终将输出1~5这5个数字。

**二、Java Stream API进阶篇**

<http://www.cnblogs.com/CarpenterLee/p/6550212.html>

**规约操作**（reduction operation）又被称作**折叠操作（fold）**，是**通过某个连接动作将所有元素汇总成一个汇总结果的过程**。元素求和、求最大值或最小值、求出元素总个数、将所有元素转换成一个列表或集合，都属于规约操作。Stream类库有两个通用的规约操作**reduce()和collect()**，也有一些为简化书写而设计的专用规约操作，比如sum()、max()、min()、count()等。

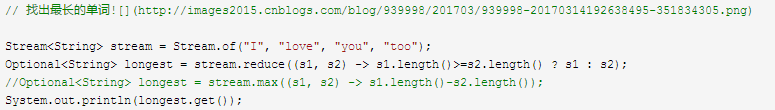
**1、多面手reduce()**

reduce操作可以实现从一组元素中生成一个值，sum()、max()、min()、count()等都是reduce操作，将他们单独设为函数只是因为常用。reduce()的方法定义有三种重写形式：

* Optional<T> reduce(**BinaryOperator**<T> accumulator)
* T reduce(T identity, BinaryOperator<T> accumulator)
* <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)

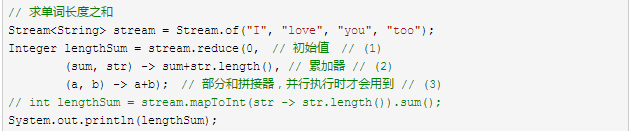
虽然函数定义越来越长，但语义不曾改变，**多的参数只是为了指明初始值（参数identity）**，或者是**指定并行执行时多个部分结果的合并方式（参数combiner）**。reduce()最常用的场景就是从一堆值中生成一个值。用这么复杂的函数去求一个最大或最小值，你是不是觉得设计者有病。其实不然，因为“大”和“小”或者“求和"有时会有不同的语义。

需求：从一组单词中找出最长的单词。这里“大”的含义就是“长”。

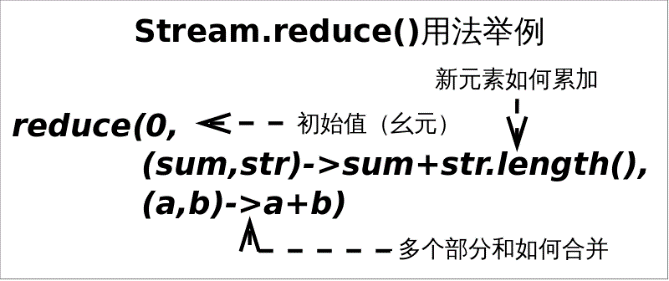


上述代码会选出最长的单词love，其中**Optional是（一个）值的容器，使用它可以避免null值的麻烦**。当然可以使用Stream.max(Comparator<? super T> comparator)方法来达到同等效果，但reduce()自有其存在的理由。

需求：求出一组单词的长度之和。这是个“求和”操作，操作对象输入类型是String，而结果类型是Integer。



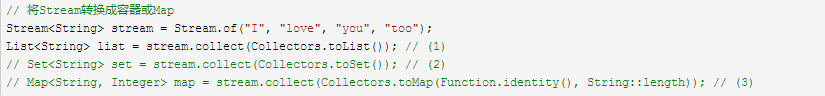
上述代码标号(2)处将**i. 字符串映射成长度，ii. 并和当前累加和相加**。这显然是两步操作，使用reduce()函数将这两步合二为一，更有助于提升性能。如果想要使用map()和sum()组合来达到上述目的，也是可以的。



reduce()擅长的是生成一个值，如果想要**从Stream生成一个集合或者Map等复杂的对象该怎么办呢？**终极武器collect()横空出世！

**2、终极武器collect()**

不夸张的讲，如果你发现某个功能在Stream接口中没找到，十有八九可以通过collect()方法实现。**collect()是Stream接口方法中最灵活的一个**，学会它才算真正入门Java函数式编程。先看几个热身的小例子：



上述代码分别列举了如何将**Stream转换成List、Set和Map**。虽然代码语义很明确，可是我们仍然会有几个疑问：

* Function.identity()是干什么的？
* String::length是什么意思？
* Collectors是个什么东西？

**1）接口的静态方法和默认方法**

Function是一个接口，那么**Function.identity()是什么意思呢？**这要从两方面解释：**Java 8允许在接口中加入具体方法**。接口中的具体方法有两种，**default方法和static方法**，**identity()就是Function接口的一个静态方法**。

Function.identity()**返回一个输出跟输入一样的Lambda表达式对象**，等价于形如t -> t形式的Lambda表达式。

上面的解释是不是让你疑问更多？不要问我为什么接口中可以有具体方法，也不要告诉我你觉得t -> t比identity()方法更直观。我会告诉你接口中的default方法是一个无奈之举，在Java 7及之前要想在定义好的接口中加入新的抽象方法是很困难甚至不可能的，因为所有实现了该接口的类都要重新实现。试想在**Collection接口中加入一个stream()抽象方法**会怎样？default方法就是用来解决这个尴尬问题的，直接在接口中实现新加入的方法。既然已经引入了default方法，为何不再加入static方法来避免专门的工具类呢！

**2）方法引用**

诸如String::length的语法形式叫做**方法引用（method references）**，这种语法用来**替代某些特定形式Lambda表达式**。如果Lambda表达式的全部内容就是**调用一个已有的方法**，那么可以用方法引用来替代Lambda表达式。方法引用可以细分为四类：

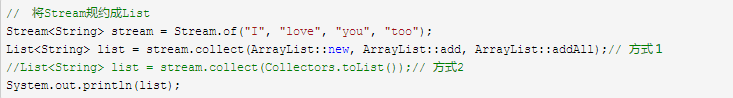


**3）收集器**

收集器（Collector）是**为Stream.collect()方法量身打造的工具接口（类）**。考虑一下将一个Stream转换成一个容器（或者Map）需要做哪些工作？我们至少需要两样东西：

* 目标容器是什么？是**ArrayList**还是**HashSet**，或者是个**TreeMap**。
* 新元素如何添加到容器中？是**List.add()还是Map.put()**。
* 如果并行的进行规约，还需要告诉collect() 3. 多个部分结果如何合并成一个。

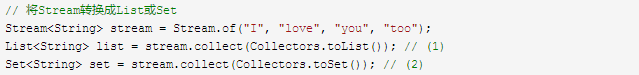
结合以上分析，collect()方法定义为**<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)**，三个参数依次对应上述三条分析。不过每次调用collect()都要传入这三个参数太麻烦，**收集器Collector就是对这三个参数的简单封装**,所以collect()的另一定义为**<R,A> R collect(Collector<? super T,A,R> collector)**。**Collectors工具类可通过静态方法生成各种常用的Collector**。举例来说，如果要将Stream规约成List可以通过如下两种方式实现：



通常情况下我们不需要手动指定collect()的三个参数，而是调用**collect(Collector<? super T,A,R> collector)方法**，并且参数中的Collector对象大都是直接通过Collectors工具类获得。**实际上传入的收集器的行为决定了collect()的行为**。

**4）使用collect()生成Collection**

前面已经提到**通过collect()方法将Stream转换成容器的方法**，这里再汇总一下。将Stream转换成List或Set是比较常见的操作，所以Collectors工具已经为我们提供了对应的收集器，通过如下代码即可完成：



上述代码能够满足大部分需求，但由于**返回结果是接口类型**，我们并不知道类库实际选择的容器类型是什么，有时候我们可能会想要**人为指定容器的实际类型**，这个需求可通过**Collectors.toCollection(Supplier<C> collectionFactory)**方法完成。



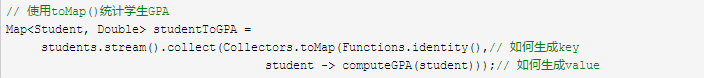
上述代码(3)处指定规约结果是ArrayList，而(4)处指定规约结果为HashSet。一切如你所愿。

**5）使用collect()生成Map**

前面已经说过Stream背后**依赖于某种数据源**，数据源可以是**数组、容器**等，但不能是Map。反过来从Stream生成Map是可以的，但我们要想清楚**Map的key和value分别代表**什么，根本原因是我们要想清楚要干什么。通常在三种情况下collect()的结果会是Map：

* 使用**Collectors.toMap()**生成的收集器，用户需要**指定如何生成Map的key和value**；
* 使用**Collectors.partitioningBy()**生成的收集器，对**元素进行二分区操作时用到**；
* 使用**Collectors.groupingBy()**生成的收集器，对**元素做group操作时用到**。

**情况1：**使用toMap()生成的收集器，这种情况是最直接的，前面例子中已提到，这是和**Collectors.toCollection()并列**的方法。如下代码展示将学生列表转换成由<学生，GPA>组成的Map。非常直观，无需多言。



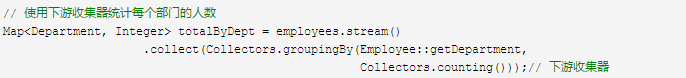
**情况2：**使用partitioningBy()生成的收集器，这种情况**适用于将Stream中的元素依据某个二值逻辑（满足条件，或不满足）分成互补相交的两部分**，比如男女性别、成绩及格与否等。下列代码展示将学生分成成绩及格或不及格的两部分。



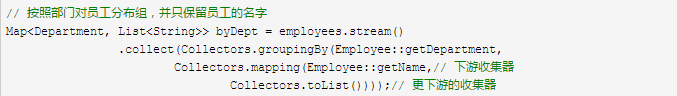
**情况3：**使用groupingBy()生成的收集器，这是比较灵活的一种情况。跟SQL中的group by语句类似，这里的**groupingBy()也是按照某个属性对数据进行分组，属性相同的元素会被对应到Map的同一个key上**。下列代码展示将员工按照部门进行分组：



以上只是分组的最基本用法，有些时候仅仅分组是不够的。在SQL中使用group by是为了协助其他查询，比如**1. 先将员工按照部门分组，2. 然后统计每个部门员工的人数**。Java类库设计者也考虑到了这种情况，增强版的groupingBy()能够满足这种需求。增强版的groupingBy()允许我们**对元素分组之后再执行某种运算，比如求和、计数、平均值、类型转换等**。这种先将元素分组的收集器叫做**上游收集器**，之后执行其他运算的收集器叫做**下游收集器**(downstream Collector)。



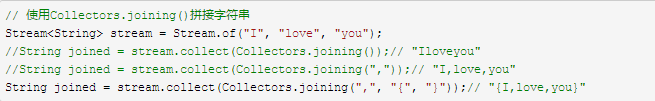
上面代码的逻辑是不是越看越像SQL？**高度非结构化**。还有更狠的，**下游收集器还可以包含更下游的收集器，这绝不是为了炫技而增加的把戏，而是实际场景需要**。考虑将员工按照部门分组的场景，如果我们想得到每个员工的名字（字符串），而不是一个个Employee对象，可通过如下方式做到：



如果看到这里你还没有对Java函数式编程失去信心，恭喜你，你已经顺利成为Java函数式编程大师了。

**6）使用collect()做字符串join**

这个肯定是大家喜闻乐见的功能，**字符串拼接时使用Collectors.joining()生成的收集器**，从此告别for循环。Collectors.joining()方法有三种重写形式，分别对应三种不同的拼接方式。无需多言，代码过目难忘。



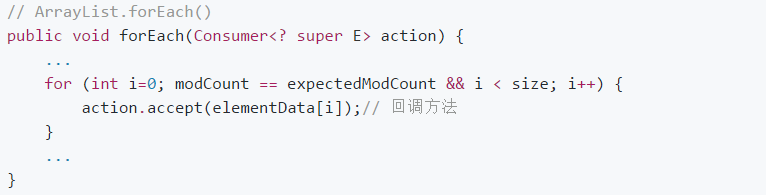
**7）collect()还可以做更多**

除了可以**使用Collectors工具类已经封装好的收集器**，我们还可以**自定义收集器**，或者**直接调用collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)方法**，收集任何形式你想要的信息。不过Collectors工具类应该能满足我们的绝大部分需求，手动实现之间请先看看文档。

**三、深入理解Java Stream流水线（PipeLines）**

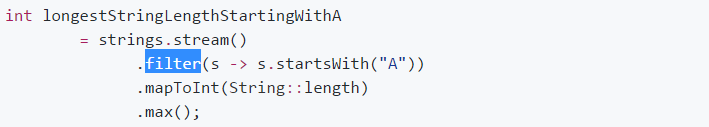
<http://www.cnblogs.com/CarpenterLee/p/6637118.html>

首先回顾一下容器执行Lambda表达式的方式，以**ArrayList.forEach()方法为例**，具体代码如下：

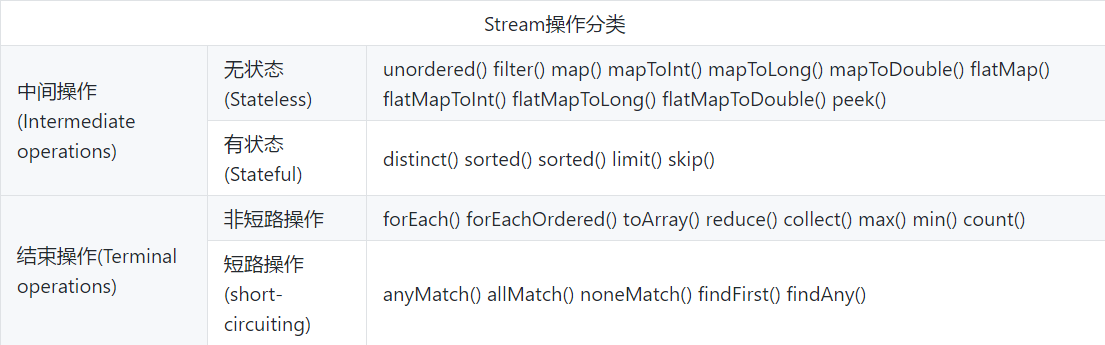


我们看到ArrayList.forEach()方法的主要逻辑就是一个for循环，在该for循环里不断调用**action.accept()回调方法完成对元素的遍历**。这完全没有什么新奇之处，回调方法在Java GUI的监听器中广泛使用。**Lambda表达式的作用就是相当于一个回调方法**，这很好理解。

Stream API中**大量使用Lambda表达式作为回调方法**，但这并不是关键。理解Stream我们更关心的是另外两个问题：**流水线和自动并行**。使用Stream或许很容易写入如下形式的代码：



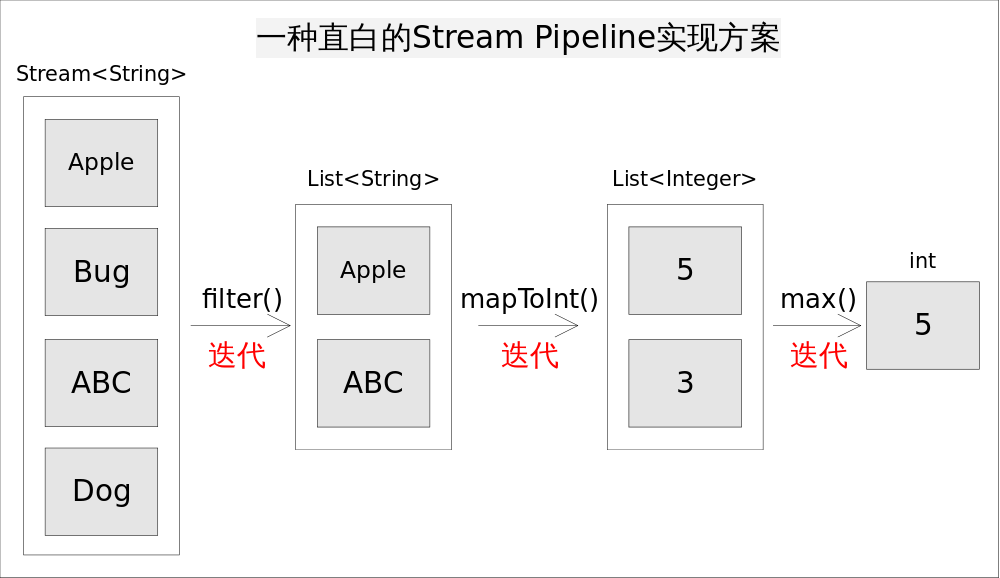
上述代码求出以字母A开头的字符串的最大长度，一种直白的方式是为每一次函数调用都执一次迭代，这样做能够实现功能，但效率上肯定是无法接受的。**类库的实现着使用流水线（Pipeline）的方式巧妙的避免了多次迭代，其基本思想是在一次迭代中尽可能多的执行用户指定的操作**。为讲解方便我们汇总了Stream的所有操作。



Stream上的所有操作分为两类：**中间操作和结束操作**，**中间操作只是一种标记，只有结束操作才会触发实际计算**。中间操作又可以分为无状态的(Stateless)和有状态的(Stateful)，无状态中间操作是**指元素的处理不受前面元素的影响**，而有状态的中间操作**必须等到所有元素处理之后才知道最终结果**，比如排序是有状态操作，在读取所有元素之前并不能确定排序结果；结束操作又可以分为短路操作和非短路操作，短路操作是指不用处理全部元素就可以返回结果，比如找到第一个满足条件的元素。之所以要进行如此精细的划分，是因为底层对每一种情况的处理方式不同。

**1、一种直白的实现方式**

仍然考虑上述求最长字符串的程序，一种**直白的流水线实现方式**是为每一次函数调用都执一次迭代，并将处理中间结果放到某种数据结构中（比如数组，容器等）。具体说来，就是调用filter()方法后立即执行，选出所有以A开头的字符串并放到一个列表list1中，之后让list1传递给mapToInt()方法并立即执行，生成的结果放到list2中，最后遍历list2找出最大的数字作为最终结果。程序的执行流程如如所示：



这样做实现起来非常简单直观，但有两个明显的弊端：

* **迭代次数多**。迭代次数跟函数调用的次数相等；
* **频繁产生中间结果**。每次函数调用都产生一次中间结果，存储开销无法接受。

这些弊端使得效率底下，根本无法接受。如果不使用Stream API我们都知道上述代码该如何在一次迭代中完成，大致是如下形式：



采用这种方式我们**不但减少了迭代次数，也避免了存储中间结果**，显然这就是流水线，因为我们把三个操作放在了一次迭代当中。只要我们事先知道用户意图，总是能够采用上述方式实现跟Stream API等价的功能，但问题是Stream类库的设计者并不知道用户的意图是什么。如何在**无法假设用户行为的前提下实现流水线**，是类库的设计者要考虑的问题。

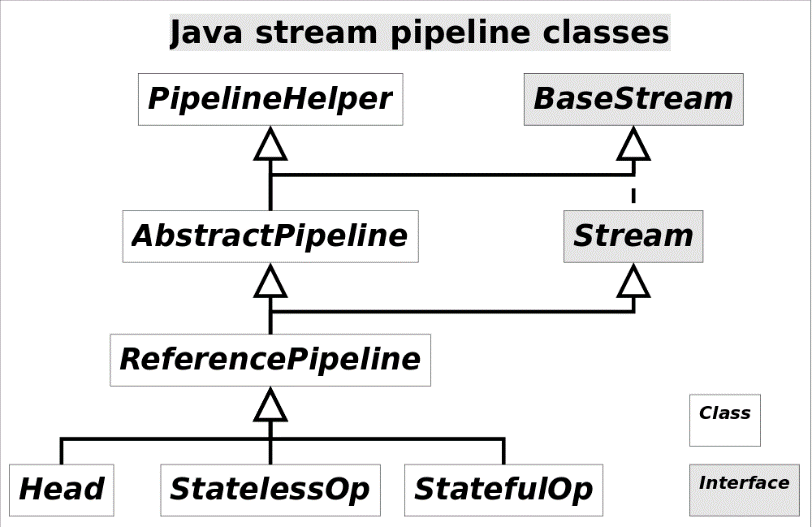
**2、Stream流水线解决方案**

我们大致能够想到，应该**采用某种方式记录用户每一步的操作**，当用户调用结束操作时将之前记录的操作叠加到一起在一次迭代中全部执行掉。沿着这个思路，有几个问题需要解决：

* 用户的操作如何记录？
* 操作如何叠加？
* 叠加之后的操作如何执行？
* 执行后的结果（如果有）在哪里？

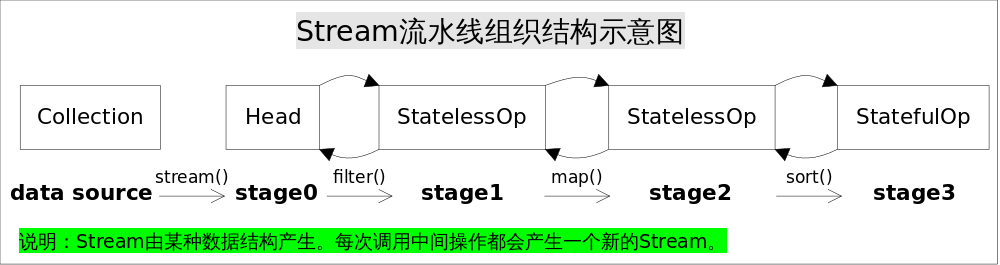
**1）操作如何记录？**

注意这里使用的是“操作(operation)”一词，指的是**“Stream中间操作”的操作**，很多Stream操作会需要一个回调函数（Lambda表达式），因此一个完整的操作是<数据来源，操作，回调函数>构成的三元组。Stream中使用Stage的概念来描述一个完整的操作，并用某种实例化后的**PipelineHelper**来代表Stage，将具有先后顺序的各个Stage连到一起，就构成了整个流水线。跟Stream相关类和接口的继承关系图示。



还有IntPipeline, LongPipeline, DoublePipeline没在图中画出，这三个类专门为三种基本类型（不是包装类型）而定制的，跟**ReferencePipeline是并列关系**。图中Head用于表示第一个Stage，即调用调用诸如Collection.stream()方法产生的Stage，很显然这个Stage里不包含任何操作；**StatelessOp和StatefulOp分别表示无状态和有状态的Stage，对应于无状态和有状态的中间操作**。

Stream流水线组织结构示意图如下：

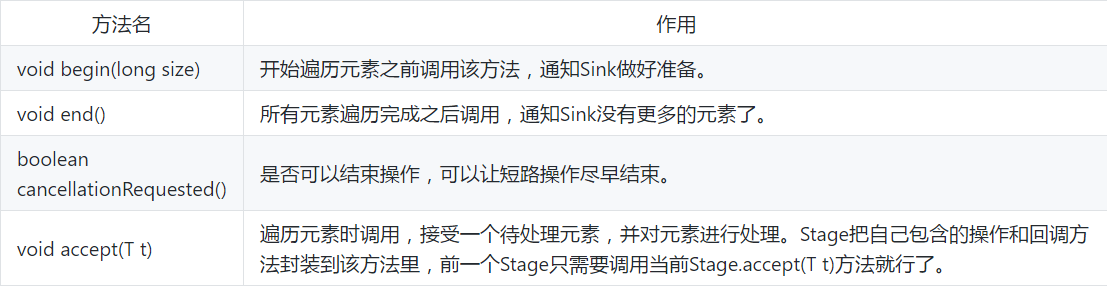


图中通过Collection.stream()方法得到**Head也就是stage0**，紧接着调用一系列的中间操作，不断产生新的Stream。这些Stream对象以**双向链表**的形式组织在一起，构成整个流水线，由于每个Stage都记录了前一个Stage和本次的操作以及回调函数，依靠这种结构就能建立起对数据源的所有操作。这就是Stream记录操作的方式。

**2）操作如何叠加？**

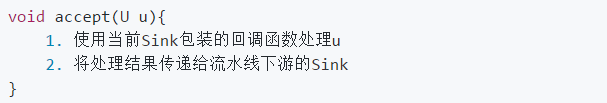
以上只是解决了操作记录的问题，要想**让流水线起到应有的作用我们需要一种将所有操作叠加到一起的方案**。你可能会觉得这很简单，只需要从流水线的head开始依次执行每一步的操作（包括回调函数）就行了。这听起来似乎是可行的，但是你忽略了**前面的Stage并不知道后面Stage到底执行了哪种操作，以及回调函数是哪种形式**。换句话说，**只有当前Stage本身才知道该如何执行自己包含的动作**。这就需要有某种协议来协调相邻Stage之间的调用关系。

这种协议由Sink接口完成，Sink接口包含的方法如下表所示：



有了上面的协议，相邻Stage之间调用就很方便了，**每个Stage都会将自己的操作封装到一个Sink里，前一个Stage只需调用后一个Stage的accept()方法即可**，并不需要知道其内部是如何处理的。当然对于**有状态的操作**，Sink的begin()和end()方法也是必须实现的。比如Stream.sorted()是一个有状态的中间操作，其对应的Sink.begin()方法可能创建一个乘放结果的容器，而accept()方法**负责将元素添加到该容器**，最后end()负责对容器进行排序。对于**短路操作**，Sink.cancellationRequested()也是必须实现的，比如Stream.findFirst()是短路操作，只要找到一个元素，cancellationRequested()就应该返回true，以便调用者尽快结束查找。Sink的四个接口方法常常相互协作，共同完成计算任务。**实际上Stream API内部实现的的本质，就是如何重载Sink的这四个接口方法**。

有了Sink对操作的包装，Stage之间的调用问题就解决了，**执行时只需要从流水线的head开始对数据源依次调用每个Stage对应的Sink.{begin(), accept(), cancellationRequested(), end()}方法就可以了**。一种可能的Sink.accept()方法流程是这样的：



Sink接口的其他几个方法也是按照这种**[处理->转发]的模型实现**。下面我们结合具体例子看看Stream的中间操作是**如何将自身的操作包装成Sink以及Sink是如何将处理结果转发给下一个Sink的**。先看**Stream.map()方法**：



上述代码看似复杂，其实逻辑很简单，就是将**回调函数mapper包装到一个Sink当中**。由于Stream.map()是一个无状态的中间操作，所以map()方法返回了一个StatelessOp内部类对象（一个新的Stream），调用这个新Stream的opWripSink()方法将得到一个包装了当前回调函数的Sink。

再来看一个复杂一点的例子。Stream.sorted()方法将对Stream中的元素进行排序，显然这是一个有状态的中间操作，因为读取所有元素之前是没法得到最终顺序的。抛开模板代码直接进入问题本质，sorted()方法是如何将操作封装成Sink的呢？sorted()一种可能封装的Sink代码如下：







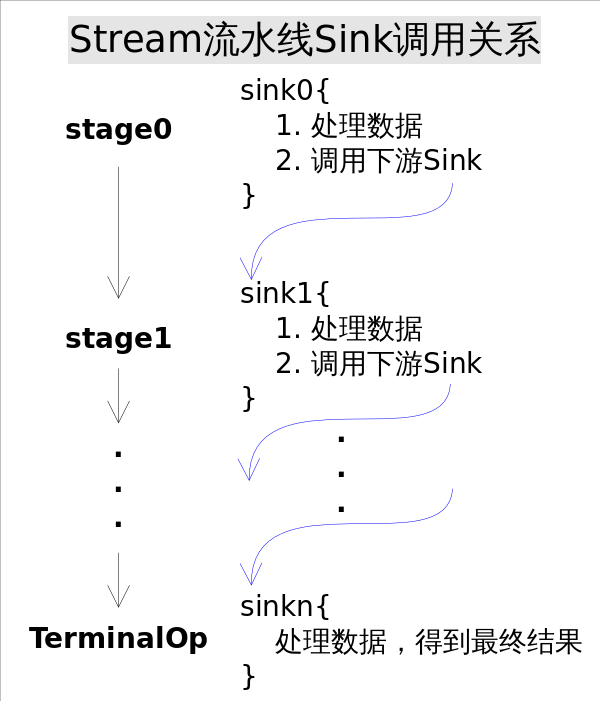
上述代码完美的展现了Sink的四个接口方法是如何协同工作的：

* 首先beging()方法告诉Sink参与排序的元素个数，方便确定中间结果容器的的大小；
* 之后**通过accept()方法将元素添加到中间结果当**中，最终执行时调用者会不断调用该方法，直到遍历所有元素；
* 最后end()方法告诉Sink所有元素遍历完毕，**启动排序步骤**，排序完成后将结果传递给下游的Sink；
* 如果下游的Sink是短路操作，将结果传递给下游时不断询问下游**cancellationRequested**()是否可以结束处理。

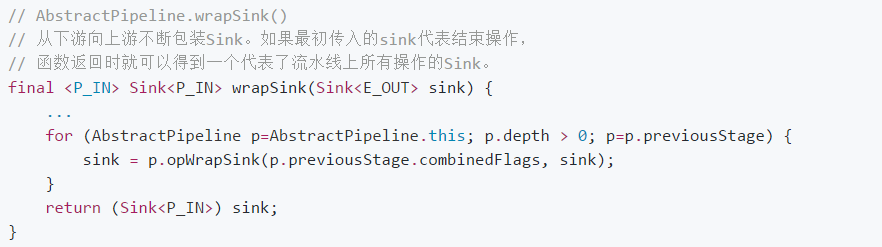
**3）叠加之后的操作如何执行？**

Sink完美封装了Stream每一步操作，并给出了**[处理->转发]的模式来叠加操作**。这一连串的齿轮已经咬合，就差最后一步拨动齿轮启动执行。是什么启动这一连串的操作呢？也许你已经想到了启动的原始动力就是结束操作(Terminal Operation)，一旦调用某个结束操作，就会触发整个流水线的执行。

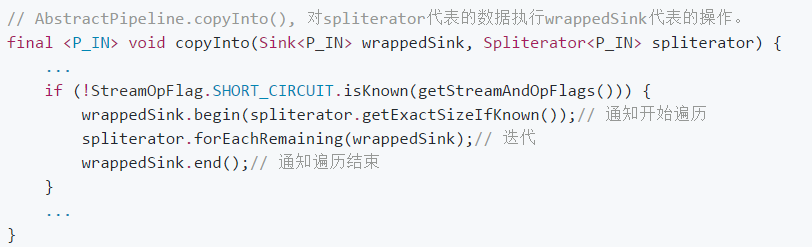
结束操作之后不能再有别的操作，所以结束操作不会创建新的流水线阶段(Stage)，直观的说就是流水线的链表不会在往后延伸了。**结束操作会创建一个包装了自己操作的Sink，这也是流水线中最后一个Sink，这个Sink只需要处理数据而不需要将结果传递给下游的Sink（因为没有下游）**。对于Sink的[处理->转发]模型，结束操作的Sink就是调用链的出口。



我们再来考察一下**上游的Sink是如何找到下游Sink的**。一种可选的方案是在PipelineHelper中设置一个Sink字段，在流水线中找到下游Stage并访问Sink字段即可。但Stream类库的设计者没有这么做，而是设置了一个**Sink AbstractPipeline.opWrapSink(int flags, Sink downstream)方法来得到Sink**，该方法的作用是**返回一个新的包含了当前Stage代表的操作以及能够将结果传递给downstream的Sink对象**。为什么要产生一个新对象而不是返回一个Sink字段？这是因为使用opWrapSink()可以将当前操作与下游Sink（上文中的downstream参数）结合成新Sink。试想只要从流水线的最后一个Stage开始，不断调用上一个Stage的opWrapSink()方法直到最开始（不包括stage0，因为stage0代表数据源，不包含操作），就可以得到一个代表了流水线上所有操作的Sink，用代码表示就是这样：



现在流水线上从开始到结束的所有的操作都被包装到了一个Sink里，执行这个Sink就相当于执行整个流水线，执行Sink的代码如下：

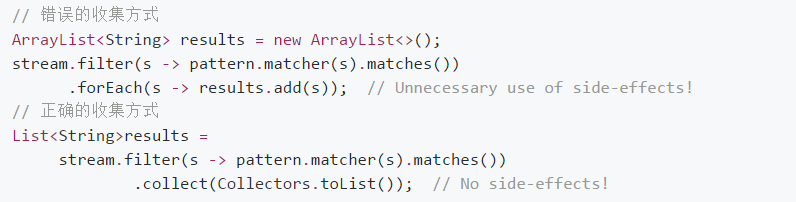


上述代码首先调用wrappedSink.begin()方法告诉Sink数据即将到来，然后调用spliterator.forEachRemaining()方法对数据进行迭代（Spliterator是容器的一种迭代器，参阅），最后调用wrappedSink.end()方法通知Sink数据处理结束。逻辑如此清晰。

**4）执行后的结果在哪里？**

最后一个问题是流水线上所有操作都执行后，用户所需要的结果（如果有）在哪里？首先要说明的是**不是所有的Stream结束操作都需要返回结果**，有些操作只是为了使用其副作用(Side-effects)，比如使用**Stream.forEach()方法将结果打印出来就是常见的使用副作用的场景**（事实上，除了打印之外其他场景都应避免使用副作用），对于真正需要返回结果的结束操作结果存在哪里呢？

**特别说明：**副作用不应该被滥用，也许你会觉得在Stream.forEach()里进行元素收集是个不错的选择，就像下面代码中那样，但遗憾的是**这样使用的正确性和效率都无法保证**，因为**Stream可能会并行执行**。大多数使用副作用的地方都可以使用**归约操作更安全和有效的完成**。



回到流水线执行结果的问题上来，需要返回结果的流水线结果存在哪里呢？这要分不同的情况讨论，下表给出了**各种有返回结果的Stream结束操作**。



* 对于表中返回**boolean或者Optional**的操作（**Optional是存放一个null值的容器**）的操作，由于值返回一个值，只需要在对应的Sink中记录这个值，等到执行结束时返回就可以了；
* 对于**归约操作**，最终结果放在**用户调用时指定的容器中（容器类型通过收集器指定）**。collect(), reduce(), max(), min()都是归约操作，虽然max()和min()也是返回一个Optional，但事实上底层是通过调用reduce()方法实现的；
* 对于返回是**数组**的情况，毫无疑问的结果会放在数组当中。这么说当然是对的，但在最终**返回数组之前，结果其实是存储在一种叫做Node的数据结构中的**。Node是一种多叉树结构，元素存储在树的叶子当中，并且一个叶子节点可以存放多个元素。这样做是为了并行执行方便。关于Node的具体结构，我们会在下一节探究Stream如何并行执行时给出详细说明。