大标题

# lambda和匿名类

## 知识储备

### 匿名内部类class文件

在JVM层面，Lambda表达式和匿名内部类有着明显的差别，匿名内部类编译之后，产生主类和匿名内部类两个class文件



进一步分析主类MainAnonymousClass.class的字节码，可发现其创建了匿名内部类的对象

// javap -c MainAnonymousClass.class

public class MainAnonymousClass {

...

public static void main(java.lang.String[]);

Code:

0: new #2 // class java/lang/Thread

3: dup

4: new #3 // class MainAnonymousClass$1 /\*创建内部类对象\*/

7: dup

8: invokespecial #4 // Method MainAnonymousClass$1."<init>":()V

11: invokespecial #5 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V

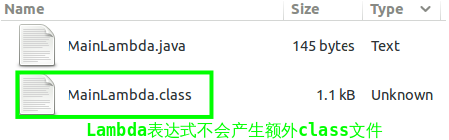
14: invokevirtual #6 // Method java/lang/Thread.start:()V

17: return

}

### invokedynamic指令不产生新的类

JVM内部是通过invokedynamic指令来实现Lambda表达式的



通过javap反编译命名，我们更能看出Lambda表达式内部表示的不同：

// javap -c -p MainLambda.class

public class MainLambda {

...

public static void main(java.lang.String[]);

Code:

0: new #2 // class java/lang/Thread

3: dup

4: invokedynamic #3, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable; /\*使用invokedynamic指令调用\*/

9: invokespecial #4 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V

12: invokevirtual #5 // Method java/lang/Thread.start:()V

15: return

private static void lambda$main$0(); /\*Lambda表达式被封装成主类的私有方法\*/

Code:

0: getstatic #6 // Field java/lang/System.out:Ljava/io/PrintStream;

3: ldc #7 // String Lambda Thread run()

5: invokevirtual #8 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

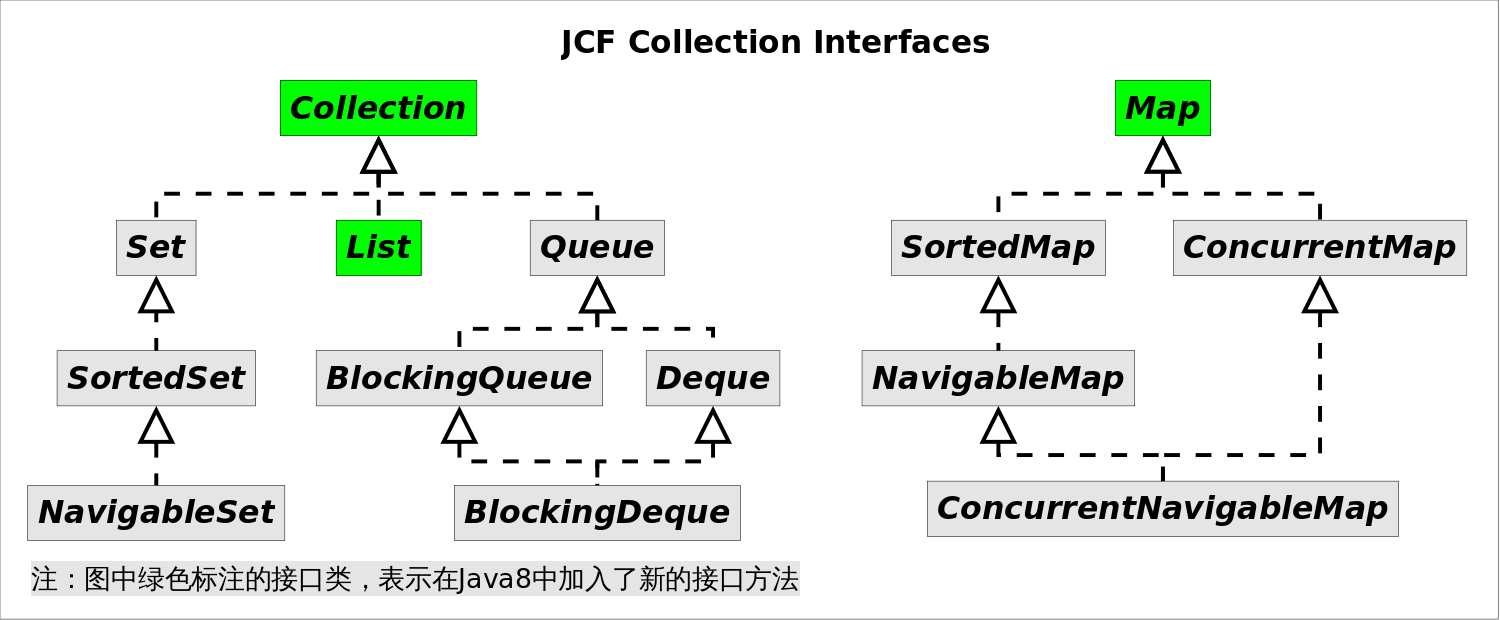
8: return

}

反编译之后我们发现Lambda表达式被封装成了主类的一个私有方法，并通过invokedynamic指令进行调用；在Lambda表达式中，this的意义跟在表达式外部完全一样

### lambda和Java集合框架

（1）Lambda表达式的基础：Java8新增了java.util.function包，里面包含常用的函数接口；（2）为了与Lambda表达式对接：Java集合框架新增部分接口（List）和方法



详见<https://github.com/962014008/JavaLambdaInternals.git>的LambdaAndAnonymousClass

### reduce操作必备：Function、BiFunction和BinaryOperator

@FunctionalInterface  
public interface Function<T, R> {  
  
 /\*\*  
 \* Applies this function to the given argument.  
 \*  
 \* **@param** t the function argument  
 \* **@return** the function result  
 \*/  
 R apply(T t);  
  
 ...  
  
 /\*\*  
 \* Returns a function that always returns its input argument.  
 \*  
 \* **@param** <T> the type of the input and output objects to the function  
 \* **@return** a function that always returns its input argument  
 \*/  
 static <T> Function<T, T> identity() {  
 return t -> t;  
 }  
}  
  
/\*\*  
 \* Represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of {**@code** Function} for the case where the operand and result are of the same type.  
 \*This is a functional interface whose functional method is {**@link** #apply(Object)}.  
 \*  
 \* **@param** <T> the type of the operand and result of the operator  
 \*  
 \* **@see** Function  
 \*/  
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
  
 /\*\*  
 \* Returns a unary operator that always returns its input argument.  
 \*  
 \* **@param** <T> the type of the input and output of the operator  
 \* **@return** a unary operator that always returns its input argument  
 \*/  
 static <T> UnaryOperator<T> identity() {  
 return t -> t;  
 }  
}

/\*\*  
 \* Represents a function that accepts two arguments and produces a result. This is the two-arity specialization of {**@link** Function}. This is a functional interface whose functional method is {**@link** #apply(Object, Object)}.  
 \*  
 \* **@param** <T> the type of the first argument to the function  
 \* **@param** <U> the type of the second argument to the function  
 \* **@param** <R> the type of the result of the function  
 \* BiFunction接受两个任意类型的参数，并返回任意类型的结果  
 \*/  
@FunctionalInterface  
public interface BiFunction<T, U, R> {  
  
 /\*\*  
 \* Applies this function to the given arguments.  
 \*  
 \* **@param** t the first function argument  
 \* **@param** u the second function argument  
 \* **@return** the function result  
 \*/  
 R apply(T t, U u);  
  
 ...  
}  
  
/\*\*  
 \* Represents an operation upon two operands of the same type, producing a result of the same type as the operands. This is a specialization of {**@link** BiFunction} for the case where the operands and the result are all of the same type.  
 \* This is a functional interface whose functional method is {**@link** #apply(Object, Object)}.  
 \*  
 \* **@param** <T> the type of the operands and result of the operator  
 \*  
 \* **@see** BiFunction  
 \* **@see** UnaryOperator  
 \* BinaryOperator接受两个相同类型的参数，并返回其参数相同类型的结果  
 \*/  
@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T,T,T> {  
 /\*\*  
 \* Returns a {**@link** BinaryOperator} which returns the lesser of two elements according to the specified {**@code** Comparator}.  
 \*  
 \* **@param** <T> the type of the input arguments of the comparator  
 \* **@param** comparator a {**@code** Comparator} for comparing the two values  
 \* **@return** a {**@code** BinaryOperator} which returns the lesser of its operands, according to the supplied {**@code** Comparator}  
 \*/  
 public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator) {  
 Objects.*requireNonNull*(comparator);  
 return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;  
 }  
  
 /\*\*  
 \* Returns a {**@link** BinaryOperator} which returns the greater of two elements according to the specified {**@code** Comparator}.  
 \*  
 \* **@param** <T> the type of the input arguments of the comparator  
 \* **@param** comparator a {**@code** Comparator} for comparing the two values  
 \* **@return** a {**@code** BinaryOperator} which returns the greater of its operands, according to the supplied {**@code** Comparator}  
 \*/  
 public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) {  
 Objects.*requireNonNull*(comparator);  
 return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;  
 }  
}

### collect操作必备：Supplier、BiConsumer

/\*\*  
 \* Represents a supplier of results.  
 \*  
 \* <p>There is no requirement that a new or distinct result be returned each  
 \* time the supplier is invoked.  
 \*  
 \* <p>This is a <a href="package-summary.html">functional interface</a>  
 \* whose functional method is {**@link** #get()}.  
 \*  
 \* **@param** <T> the type of results supplied by this supplier  
 \*  
 \* **@since** 1.8  
 \*/  
@FunctionalInterface  
public interface Supplier<T> {  
  
 /\*\*  
 \* Gets a result.  
 \*  
 \* **@return** a result  
 \*/  
 T get();  
}  
  
/\*\*  
 \* Represents an operation that accepts two input arguments and returns no  
 \* result. This is the two-arity specialization of {**@link** Consumer}.  
 \* Unlike most other functional interfaces, {**@code** BiConsumer} is expected  
 \* to operate via side-effects.  
 \*  
 \* <p>This is a <a href="package-summary.html">functional interface</a>  
 \* whose functional method is {**@link** #accept(Object, Object)}.  
 \*  
 \* **@param** <T> the type of the first argument to the operation  
 \* **@param** <U> the type of the second argument to the operation  
 \*  
 \* **@see** Consumer  
 \* **@since** 1.8  
 \*/  
@FunctionalInterface  
public interface BiConsumer<T, U> {  
  
 /\*\*  
 \* Performs this operation on the given arguments.  
 \*  
 \* **@param** t the first input argument  
 \* **@param** u the second input argument  
 \*/  
 void accept(T t, U u);  
  
 /\*\*  
 \* Returns a composed {**@code** BiConsumer} that performs, in sequence, this  
 \* operation followed by the {**@code** after} operation. If performing either  
 \* operation throws an exception, it is relayed to the caller of the  
 \* composed operation. If performing this operation throws an exception,  
 \* the {**@code** after} operation will not be performed.  
 \*  
 \* **@param** after the operation to perform after this operation  
 \* **@return** a composed {**@code** BiConsumer} that performs in sequence this  
 \* operation followed by the {**@code** after} operation  
 \* **@throws** NullPointerException if {**@code** after} is null  
 \*/  
 default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {  
 Objects.*requireNonNull*(after);  
  
 return (l, r) -> {  
 accept(l, r);  
 after.accept(l, r);  
 };  
 }  
}

## Java 8新增接口（List）和方法

Collection：removeIf() spliterator() stream() parallelStream() forEach()

List：replaceAll() sort()

Map：getOrDefault() forEach() replaceAll() putIfAbsent() remove() replace() computeIfAbsent() computeIfPresent() compute() merge()

这些新加入的方法大部分要用到java.util.function包下的接口，详见<https://github.com/962014008/JavaLambdaInternals.git>的NewInterfaceAndFunction，分类如下：

// 1.Collection接口（Set、List和Queue的父接口）  
// 1.1 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，其中Consumer是个函数接口，里面只有一个待实现方法void accept(T t)  
// 需求：假设有一个字符串列表，需要打印出其中所有长度大于3的字符串

// 1.2 boolean removeIf(Predicate<? super E> filter)，作用是删除容器中所有满足filter指定条件的元素，其中Predicate是一个函数接口，里面只有一个待实现方法boolean test(T t)  
// 需求：假设有一个字符串列表，需要删除其中所有长度大于3的字符串

// 1.3 Spliterator<E> spliterator()，该方法返回容器的可拆分迭代器，该方法跟iterator()方法有点像，Spliterator也有类似迭代容器的作用，但二者有如下不同  
// (1) Spliterator既可以像Iterator那样逐个迭代，也可以批量迭代，批量迭代可以降低迭代的开销  
// (2) Spliterator可拆分，可以通过Spliterator<T> trySplit()方法来尝试分成两个迭代器，一个是this，另一个是新返回的（元素无重叠）；可多次调用trySplit()方法来分解负载，以便多线程处理  
  
// 1.4 stream()和parallelStream()，分别返回容器的Stream视图表示，不同点在于parallelStream()返回并行的Stream，Stream是Java函数式编程的核心类  
  
// 2.List接口  
// 2.1 void replaceAll(UnaryOperator<E> operator)，作用是对每个元素执行operator指定的操作，并用操作结果来替换原来的元素，其中UnaryOperator是一个函数接口，里面只有一个待实现函数T apply(T t)  
// 需求：假设有一个字符串列表，将其中所有长度大于3的元素转换成大写，其余元素不变

// 2.2 void sort(Comparator<? super E> c)，该方法根据c指定的比较规则对容器元素进行排序。Comparator接口我们并不陌生，其中有一个方法int compare(T o1, T o2)需要实现，显然该接口是个函数接口  
// 需求：假设有一个字符串列表，按照字符串长度增序对元素排序

// 3.Map接口  
// 3.1 void forEach(BiConsumer<? super K,? super V> action)，作用是对Map中的每个映射执行action指定的操作，其中BiConsumer是一个函数接口，里面有一个待实现方法void accept(T t, U u)  
// 需求：假设有一个数字到对应英文单词的Map，请输出Map中的所有映射关系

// 3.2 V getOrDefault(Object key, V defaultValue)，按照给定的key查询Map中对应的value，如果没有找到则返回defaultValue；该方法跟Lambda表达式没关系，但是很有用  
// 需求：假设有一个数字到对应英文单词的Map，输出4对应的英文单词，如果不存在则输出NoValue

// 3.3 V putIfAbsent(K key, V value)，将条件判断和赋值合二为一，仅当key的映射值不存在或为null时，才将value值放入Map中，否则不做更改；该方法跟Lambda表达式没关系，但是很有用  
// 3.4 remove(Object key, Object value)，Java8新增，仅当Map中key正好映射到value时才删除该映射，否则什么也不做；Map在Java7中已经有一个remove(Object key)方法，删除Map中指定key的映射关系  
// 3.5 replace方法；Map在Java7中，要可通过put(K key, V value) 方法替换映射值，该方法总是会用新值替换原来的值，为了更精确的控制替换行为，Java8加入了两个replace方法  
// (1) V replace(K key, V value)，仅当Map中key的映射存在时才用value去替换原来的值，否则什么也不做   
// (2) boolean replace(K key, V oldValue, V newValue)，仅当Map中key的映射存在且等于oldValue时，才用newValue去替换原来的值，否则什么也不做  
// 3.6 void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)  
// 作用是对Map中的每个映射执行function指定的操作，并用function的执行结果替换原来的value；其中BiFunction是一个函数接口，里面有一个待实现方法R apply (K k, V v)  
// 需求：假设有一个数字到对应英文单词的Map，请将原来映射关系中的单词都转换成大写

// 3.7 V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)  
// (1) 如果Map中key对应的映射不存在或者为null时，将value（不能是null）关联到key上  
// (2) 否则执行remappingFunction，如果执行结果非null，则用该结果跟key关联，否则在Map中删除key的映射；其中BiFunction是一个函数接口，里面有一个待实现方法R apply (K k, V v)  
// 需求：将新的错误信息拼接到原来的信息上

// 3.8 V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)，如果计算结果为null，则在Map中删除key的映射，否则把remappingFunction的计算结果跟key关联  
// 需求：将新的错误信息拼接到原来的信息上

// 3.9 V computeIfAbsent (K key, Function<? super K, ? extends V> mappingFunction)，将条件判断和添加操作合二为一  
// 仅当Map中key值的映射不存在或为null时，才调用mappingFunction，并在mappingFunction执行结果非null时，将结果跟key关联；Function是一个函数接口，里面有一个待实现方法R apply (T t)  
// 需求：对Map的某个key值建立初始化映射．比如我们要实现一个多值映射，Map的定义可能是Map<K, Set<V>>，要向Map中放入新值

// 3.10 V computeIfPresent (K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)；其中BiFunction是一个函数接口，里面有一个待实现方法R apply (K k, V v)  
// 仅当Map中存在key值的映射且非null时，才调用remappingFunction，如果remappingFunction执行结果为null，则删除key的映射，否则使用该结果替换key原来的映射

# Stream API

## 基本概念

### Java 8引入函数式编程的原因

（1）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

以ArrayList.forEach()方法为例，具体代码如下：

// ArrayList.forEach()

public void forEach(Consumer<? super E> action) {

...

for (int i=0; modCount == expectedModCount && i < size; i++) {

action.accept(elementData[i]);// 回调方法

}

...

}

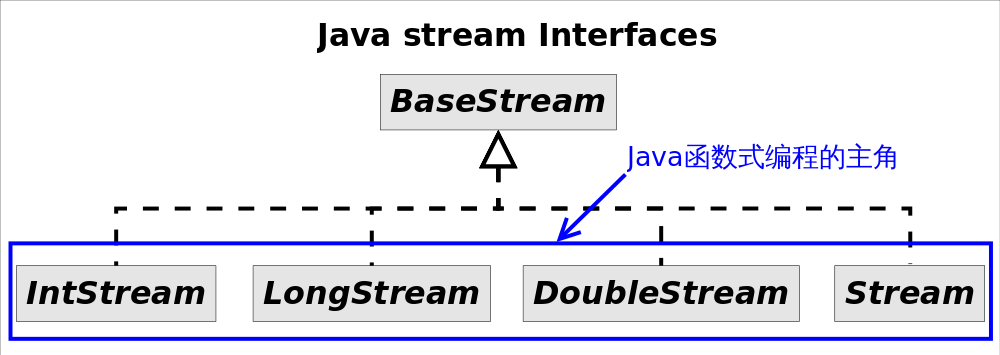
主要逻辑是一个for循环，不断调用action.accept()回调方法完成对元素的遍历，这里Lambda表达式的作用就是相当于一个回调方法

（2）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是Java函数式编程的主角

stream并不是某种数据结构，而是数据源的一种视图（数据源可以是一个数组、Java容器或I/O channel等，但不能是Map），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

## stream接口继承关系



### 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

### 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：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

## stream和collections的异同

### 相同

大部分情况下，stream是容器调用Collection.stream()方法得到的

### 差异

（1）无存储。stream不是一种数据结构，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；

（2）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；

（3）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；

（4）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

## stream操作

### 对stream的操作分为两类

中间操作(intermediate operations)和结束操作(terminal operations)，二者特点是：1.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；2.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

### Stream接口的部分常见方法

操作类型 接口方法

中间操作 concat() distinct() filter() flatMap() limit() map() peek() skip() sorted() parallel() sequential() unordered()

结束操作 allMatch() anyMatch() collect() count() findAny() findFirst() forEach() forEachOrdered() max() min() noneMatch() reduce() toArray()

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常用操作

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，详见<https://github.com/962014008/JavaLambdaInternals.git>的CommonStreamOperation

// 1.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；forEach()是结束操作，代码会立即执行  
// 需求：使用Stream.forEach()迭代

// 2.Stream<T> filter(Predicate<? super T> predicate)，作用是返回一个只包含满足predicate条件元素的Stream；filter()是中间操作，不会有实际计算  
// 需求：保留长度等于3的字符串

// 3.Stream<T> distinct()，作用是返回一个去除重复元素之后的Stream；distinct()是中间操作，不会有实际计算  
// 需求：去除重复元素

// 4.Stream<T> sorted()和Stream<T> sorted(Comparator<? super T> comparator)，分别是自然顺序排序（升序）、自定义比较器排序

// 5.<R> Stream<R> map(Function<? super T, ? extends R> mapper)，返回一个对当前所有元素执行mapper之后的结果组成的Stream，即对每个元素按照某种操作进行转换  
// 转换前后Stream中元素个数不会改变，但元素类型取决于转换之后的类型；map()是中间操作，不会有实际计算  
// 需求：将String数组的元素都转换成大写形式

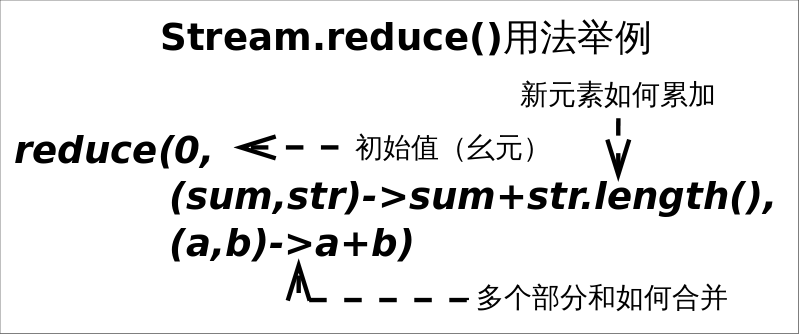
// 6.<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)，对每个元素执行mapper指定的操作，并用所有mapper返回的Stream中的元素，组成一个新的Stream作为最终返回结果  
// 即原stream中的所有元素都 "摊平" 之后组成的Stream，转换前后元素的个数和类型都可能会改变；flatMap()是中间操作，不会有实际计算  
// 需求：将List<Integer>数组的元素都“摊平”成一个新的Integer数组，输出两个List<Integer>里面的所有数字，包括1~5  
// 转换对象是List<Integer>...（注意和5.2对比，源码实现里面调用的是apply{foreach}）

## stream规约操作（折叠操作）之reduce

### 基本概念

规约操作（reduction operation）又称折叠操作（fold），通过某个连接动作将所有元素汇总成一个汇总结果；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，这里着重介绍reduce()和collect()

### 例子



其他详见<https://github.com/962014008/JavaLambdaInternals.git的ReductionStreamOperation>

## stream规约操作（折叠操作）之collect

### Function接口的静态和默认方法

（1） Java 8允许在接口中加入具体方法，包括default方法和static方法两类

（2）Function接口的一个静态方法就是identity()，该方法返回一个输出跟输入一样的Lambda表达式对象，等价于t -> t形式的Lambda表达式；引入identity方法的原因：在Java 7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，而Java 8的default方法就是用来解决这个尴尬问题的，允许直接在接口中实现新加入的方法；

（3）此外，引入static方法可以避免专门的工具类

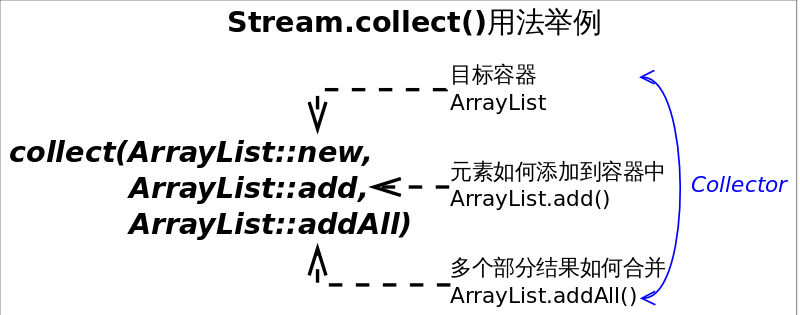
### 方法引用

（1） 基本概念：诸如String::length的语法形式叫做方法引用（method references），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。

（2）四种分类：引用静态方法，如Integer::sum；引用某个对象的方法，如list::add；引用某个类的方法，如String::length；引用构造方法，如HashMap::new

### 收集器

基本概念：为Stream.collect()方法量身打造的工具接口（类），能够将一个Stream转换成一个容器（List或者Set），或者Map，还能够并行进行规约，将多个部分结果进行合并；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，收集任何形式的信息



其他详见<https://github.com/962014008/JavaLambdaInternals.git的CollectorStreamOperation>

# Stream Pipelines

回顾前面2.1.1(1)Lambda表达式相当于回调方法，但是更关键的两个问题是流水线和自动并行

## stream的所有操作

### 操作分类

（1）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

（2）中间操作是懒惰的，不会对数据做任何操作，直到遇到了最终操作

（3）最终操作是热情的，会往前回溯所有的中间操作；当执行到最后的forEach操作，会回溯到它的上一步中间操作；上一步中间操作，又会回溯到上上一步的中间操作，直到最初的第一步

### 中间操作(Intermediate operations)

无状态(Stateless)：unordered() filter() map() mapToInt() mapToLong() mapToDouble() flatMap() flatMapToInt() flatMapToLong() flatMapToDouble() peek()

有状态(Stateful)：distinct() sorted() sorted() limit() skip()

说明：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(Terminal operations)

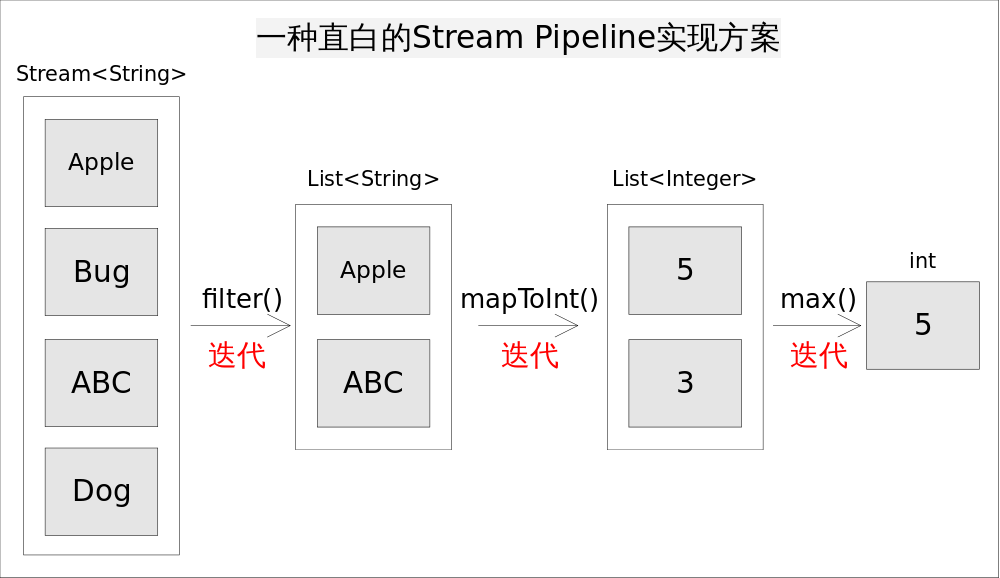
短路操作(short-circuiting)：anyMatch() allMatch() noneMatch() findFirst() findAny()

非短路操作：forEach() forEachOrdered() toArray() reduce() collect() max() min() count()

说明：分为短路操作和非短路操作，短路操作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，需要这样精细划分，是因为底层对每种情况的处理方式不同

## Stream Pipeline实现方案

### 直白实现方案（不推荐）

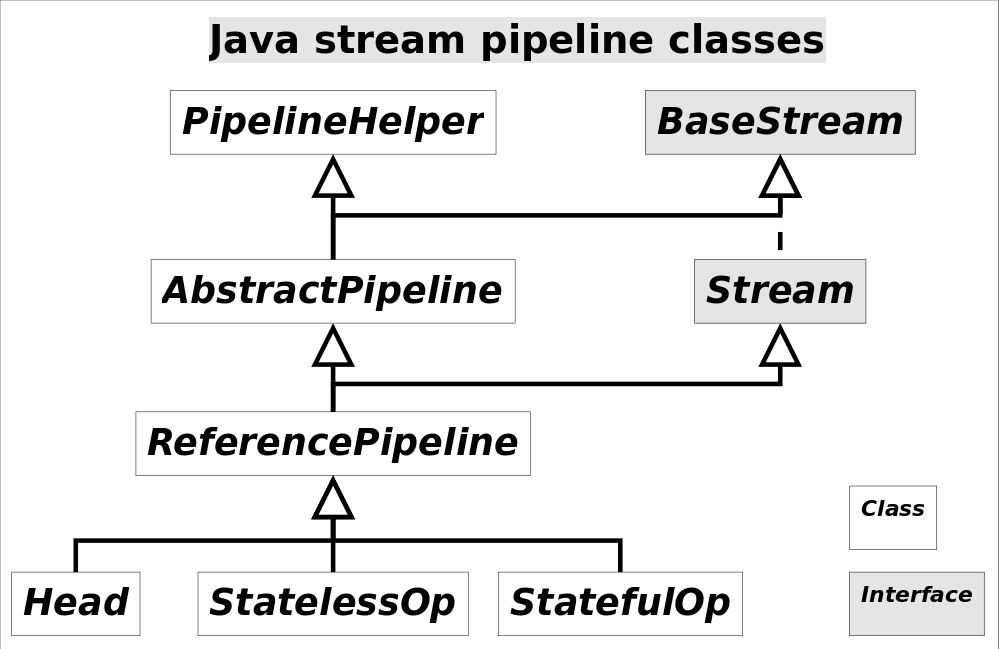


为每一次函数调用都执行一次迭代，并将处理中间结果放到某种数据结构中（比如数组，容器等）；即调用filter()方法后立即执行，选出所有以A开头的字符串并放到一个列表list1中，之后让list1传递给mapToInt()方法并立即执行，生成的结果放到list2中，最后遍历list2找出最大的数字作为最终结果；实现虽然简单，但是有两个弊端：1.迭代次数跟函数调用的次数相等，迭代次数过多；2.每次函数调用都产生一次中间结果，存储开销过大

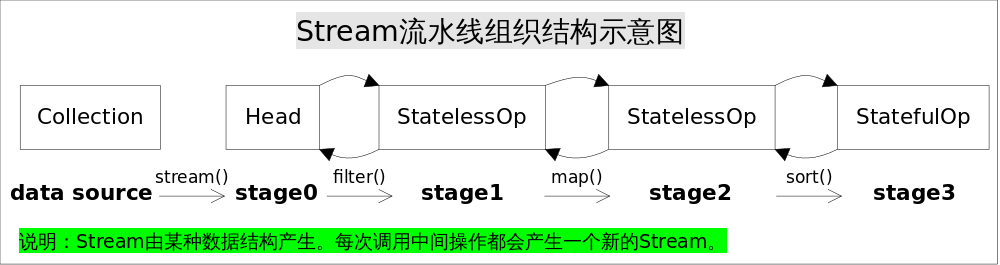
### 最终解决方案（官方）

#### 用户的操作如何记录

这里的“操作(operation)”指的是“Stream中间操作”，而操作(operation)需要一个回调函数（Lambda表达式），因此一个完整的Stream操作=一个Stage=一个由<数据来源，操作，回调函数>构成的三元组；Stream使用某种实例化后的PipelineHelper来代表Stage，将具有先后顺序的各个Stage连到一起，构成了整个流水线；跟Stream相关类和接口的继承关系如下图



还有IntPipeline, LongPipeline, DoublePipeline在图中没有画出，专门为三种基本类型（非包装类型）而定制，跟ReferencePipeline是并列关系；图中Head用于表示第一个Stage，即调用诸如Collection.stream()方法产生的Stage，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



Stream记录操作的方式：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，每个Stage都记录了前一个Stage和本次的操作以及回调函数

#### 记录好的操作如何叠加？

（1）Sink接口的作用：流水线中，前面的Stage并不知道后面Stage执行了哪种操作，以及回调函数是哪种形式；因此需要Sink接口，实现某种协议，来协调相邻Stage之间的调用关系，避免只有当前Stage本身才知道该如何执行自己包含动作的情况，Sink的四个接口方法相互协作，共同完成计算任务，如下

方法名 作用

void begin(long size) 开始遍历元素之前调用该方法，通知Sink做好准备

void end() 所有元素遍历完成之后调用，通知Sink没有更多的元素了

boolean cancellationRequested() 是否可以结束操作，可以让短路操作尽早结束

void accept(T t) 遍历元素时调用，接受一个待处理元素，并进行处理；Stage把自己包含的操作和回调方法封装到该方法，然后前一个Stage需要调用后一个Stage的accept()方法

（2）[处理->转发]的模型：Sink接口的方法都采用这种模型，从流水线的head开始，对每个Stage对应的Sink.{begin(), accept(), cancellationRequested(), end()}，依次调用

（3）对于有状态的操作：Sink的begin()和end()方法是必须实现的；比如Stream.sorted()是一个有状态的中间操作，其对应的Sink.begin()方法可能创建一个盛放结果的容器，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

（4）对于短路操作：Sink.cancellationRequested()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

（5）Sink.accept()的通用处理：

void accept(U u){

1. 使用当前Sink包装的回调函数处理u

2. 将处理结果传递给流水线下游的Sink

}

（6）Stream.map()的Sink.accept()处理：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，并传递给流水线下游

// 调用Stream.map()将产生一个新的Stream

public final <R> Stream<R> map(Function<? super P\_OUT, ? extends R> mapper) {

...

return new StatelessOp<P\_OUT, R>(this, StreamShape.REFERENCE, StreamOpFlag.NOT\_SORTED | StreamOpFlag.NOT\_DISTINCT) {

/\*opWripSink()方法返回由回调函数包装而成的Sink\*/

@Override

Sink<P\_OUT> opWrapSink(int flags, Sink<R> downstream) {

return new Sink.ChainedReference<P\_OUT, R>(downstream) {

@Override

public void accept(P\_OUT u) {

R r = mapper.apply(u);// 1. 使用当前Sink包装的回调函数mapper处理u

downstream.accept(r);// 2. 将处理结果传递给流水线下游的Sink

}

};

}

};

}

（6）Stream.sorted()的Sink.accept()处理：对Stream中的元素进行排序，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，需要Sink接口的4个方法协同完成

// Stream.sort()方法用到的Sink实现

class RefSortingSink<T> extends AbstractRefSortingSink<T> {

private ArrayList<T> list;// 存放用于排序的元素

RefSortingSink(Sink<? super T> downstream, Comparator<? super T> comparator) {

super(downstream, comparator);

}

@Override

public void begin(long size) {

...

// 创建一个存放排序元素的列表

list = (size >= 0) ? new ArrayList<T>((int) size) : new ArrayList<T>();

}

@Override

public void accept(T t) {

list.add(t);// 1. 使用当前Sink包装动作处理t，只是简单的将元素添加到中间列表当中

}

@Override

public void end() {

list.sort(comparator);// 只有元素全部接收之后才能开始排序

downstream.begin(list.size());

if (!cancellationWasRequested) {// 下游Sink不包含短路操作

list.forEach(downstream::accept);// 2. 将处理结果传递给流水线下游的Sink

}

else {// 下游Sink包含短路操作

for (T t : list) {// 每次都调用cancellationRequested()询问是否可以结束处理

if (downstream.cancellationRequested()) break;

downstream.accept(t);// 2. 将处理结果传递给流水线下游的Sink

}

}

downstream.end();

list = null;

}

}

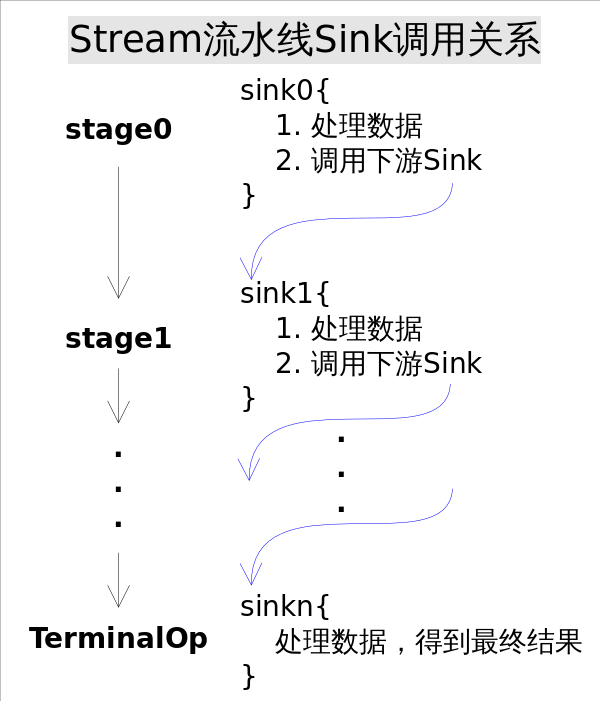
1.首先begin()方法告诉Sink参与排序的元素个数，方便确定中间结果容器的的大小；

2.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；

3.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；

4.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。

#### 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：

// AbstractPipeline.wrapSink()

// 从下游向上游不断包装Sink。如果最初传入的sink代表结束操作，

// 函数返回时就可以得到一个代表了流水线上所有操作的Sink。

final <P\_IN> Sink<P\_IN> wrapSink(Sink<E\_OUT> sink) {

...

for (AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p=p.previousStage) {

sink = p.opWrapSink(p.previousStage.combinedFlags, sink);

}

return (Sink<P\_IN>) sink;

}

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：

// AbstractPipeline.copyInto(), 对spliterator代表的数据执行wrappedSink代表的操作。

final <P\_IN> void copyInto(Sink<P\_IN> wrappedSink, Spliterator<P\_IN> spliterator) {

...

if (!StreamOpFlag.SHORT\_CIRCUIT.isKnown(getStreamAndOpFlags())) {

wrappedSink.begin(spliterator.getExactSizeIfKnown());// 通知开始遍历

spliterator.forEachRemaining(wrappedSink);// 迭代

wrappedSink.end();// 通知遍历结束

}

...

}

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。

#### 执行后的结果（如果有）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。

// 错误的收集方式

ArrayList<String> results = new ArrayList<>();

stream.filter(s -> pattern.matcher(s).matches())

.forEach(s -> results.add(s)); // Unnecessary use of side-effects!

// 正确的收集方式

List<String>results =

stream.filter(s -> pattern.matcher(s).matches())

.collect(Collectors.toList()); // No side-effects!

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。

返回类型 对应的结束操作

boolean anyMatch() allMatch() noneMatch()

Optional findFirst() findAny()

归约结果 reduce() collect()

数组 toArray()

1.对于表中返回boolean或者Optional的操作（Optional是存放 一个 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。

2.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。

3.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。

结语

本文详细介绍了Stream流水线的组织方式和执行过程，学习本文将有助于理解原理并写出正确的Stream代码，同时打消你对Stream API效率方面的顾虑。如你所见，Stream API实现如此巧妙，即使我们使用外部迭代手动编写等价代码，也未必更加高效。