关于Java Lambda表达式看这一篇就够了

2019-03-04 | 张爽爽 | Java | Lambda Collections StreamAPI

Java Lambda表达式的一个重要用法是匿名类的替代（`Anonymous Classes`）的写法。实际上Lambda表达式并不仅仅是匿名内部类的语法糖，JVM内部是通过*invokedynamic*指令来实现Lambda表达式的，具体原理放到下一篇。本篇我打算先带领一下使用Lambda表达式带来的便利之处。

Lambda and Anonymous Classes(I)

本节将介绍如何使用Lambda表达式简化匿名内部类的书写，但Lambda表达式并不能取代所有的匿名内部类，只能用来取代**函数接口**（**Functional Interface**）的简写。先到在乎细节，看几个例子再说。

例子1：无参函数的简写

如果需要新建一个线程，一种常见的写法是这样：

```
// JDK7 匿名内部类写法
new Thread(new Runnable(){// 接口名
    @Override
    public void run(){// 方法名
        System.out.println("Thread run()");
    }
}).start();
```

上述代码给 `Thread` 类传递了一个匿名的 `Runnable` 对象，重载 `Runnable` 接口的 `run()` 方法来实现相应逻辑。这是JDK7以及之前的常见写法，匿名内部类省去了为类起名字的烦恼，但还是不够简化。在Java 8中可以简化为如下形式：

```
// JDK8 Lambda表达式写法
new Thread(
        () -> {
            System.out.println("Thread run()");// 编译器以名称对应法则
        }
).start();
```

上述代码跟匿名内部类的作用是一样的，但比匿名内部类更进一步，这里连接口**名称函数名都一同省掉**了，写起来更加神清气爽。如果函数体有多行，可以用大括号括起来，就像这样：

```
// JDK8 Lambda表达式代码块写法
new Thread(
        () -> {
            System.out.print("Hello");
            System.out.println(" Hooloo");
        }
).start();
```

例子2：带参函数的简写

如果要给一个字符串列表通过自定义比较器，按照字符串长度进行排序，Java 7的书写形式如下：

```
// JDK7 匿名内部类写法
List<String> list = Arrays.asList("I", "love", "you", "too");
Collections.sort(list, new Comparator<String>(){// 接口名
    @Override
    public int compare(String s1, String s2){// 方法名
        if(s1 == null)
            return -1;
        if(s2 == null)
            return 1;
        return s1.length()-s2.length();
    }
});
```

上述代码跟匿名内部类的作用是一样的。除了省略了接口名和方法名，代码中把参数表的类型也省略了，这得益于 `javac` 的**类型推断**机制，编译器能够根据上下文信息推断出参数的类型，当然也有推断失败的时候，这时就需要手动指明参数类型了。注意，Java是强类型语言，每个变量和对象都必须有明确的类型。

简写的依据

也许你已经想到了，**能够使用Lambda的依据是必须有相应的函数接口**（函数接口，是指内部只有一个抽象方法的接口），这一点跟Java是强类型语言密切，也就是说你并不能在代码的任何地方任性的写Lambda表达式，实际上*Lambda的类型就是对应函数接口的类型*，**Lambda表达式另一个依据是类型推断机制**，在上下文信息足够的情况下，编译器可以推断出参数的类型，而不需要显式指名，Lambda表达式多余信息的书写形式如下：

```
// JDK8 Lambda表达式写法变体
Runnable run = () -> System.out.println("Hello World");// 1
ActionListener listener = event -> System.out.println("button clicked");// 2
Runnable multiline = () -> {// 3 代码块
    System.out.print("Hello");
    System.out.println(" Hooloo");
};
BinaryOperator<Long> add = (long x, long y) -> x + y;// 4
BinaryOperator<Long> addImplicit = (x, y) -> x + y;// 5 类型推断
```

上述代码中，1展示了无参函数的简写；2比展示了有参函数的简写，以及类型推断机制；3是代码块的写法；4和5再次展示了类型推断机制。

自定义函数接口

自定义函数接口很容易，只需要编写一个只有一个抽象方法的接口即可。

```
// 自定义函数接口
@FunctionalInterface
public interface ConsumerInterface-T-{
    void accept(T t);
}
```

上面代码中的@FunctionalInterface是可选的，但加上该标注编译器会帮你检查接口是否符合函数接口规范，就像加入@Override标注会检查是否重载了函数一样。

有了上述接口定义，就可以写出类似如下的代码：

```
ConsumerInterface<String> consumer = str -> System.out.println(str);
```

进一步的，还可以这样使用：

```
class MyStream-T-{
        private List-T- list;

        public void myForEach(ConsumerInterface-T- consumer){// 1
                for(T t : list){
                        consumer.accept(t);
                }
        }
}
MyStream<String> stream = new MyStream<String>();
stream.myForEach(str -> System.out.println(str));// 使用自定义函数接口书写Lambda表达式
```

Lambda and Anonymous Classes(II)

读过上一篇之后，相信对Lambda表达式的语法以及基本原理有了一定了解。对于编写代码，有这些知识已经够用。本文将**进一步区分Lambda表达式和匿名内部类在JVM层面的区别，如果对这一部分不感兴趣，可以跳过**。

经过第一篇的介绍，我们看到Lambda表达式似乎只是为了简化匿名内部类书写，这看起来仅仅通过语法糖在编译阶段把所有的Lambda表达式替换成匿名内部类就可以了，但实时并非如此。在JVM层面，Lambda表达式和匿名内部类有着明显的差别。

匿名内部类实现

匿名内部类的作用是一个类，只是不需要程序员显示指定类名，编译器会自动为该类取名，因此如果有如下形式的代码，编译之后将会产生两个class文件：

```
public class MainAnonymousClass {
    public static void main(String[] args) {
        new Thread(new Runnable(){
            @Override
            public void run(){
                System.out.println("Anonymous Class Thread run()");
            }
        }).start();;
    }
}
```

编译之后文件分书如下，两个class文件分别是主类和匿名内部类产生的：

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 /*创建内部匿名类*/
       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
}
```

Lambda表达式实现

Lambda表达式通过*invokedynamic*指令实现，书写**Lambda表达式不会产生新的类**。如果有如下形式的代码，编译之后只有一个class文件：

```
public class MainLambda {
    public static void main(String[] args) {
        new Thread(
                () -> System.out.println("Lambda Thread run()")
                ).start();;
    }
}
```

编译之后的结果：

通过javap反编译命令，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 lambdaImpl0(); /*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。

推论，this引用的意义

既然Lambda表达式不是内部类的简写，那么Lambda内部的 `this` 引用也就跟内部类时意味没什么关系了。在Lambda表达式中 `this` 的意义跟在表达式外部完全一样。因此下列代码将输出两遍 `Hello Hoolee` ，而不是两个引用地址。

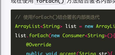
```
public class Hello {
        Runnable r1 = () -> { System.out.println(this); };
        Runnable r2 = () -> { System.out.println(toString()); };
        public static void main(String[] args) {
                new Hello().r1.run();
                new Hello().r2.run();
        }
        public String toString() { return "Hello Hoolee"; }
}
```

Lambda and Collections

我们先从最熟悉的Java集合框架Java Collections Framework, JCF开始说起。

为引入Lambda表达式，Java8新增了 `java.util.function` 包，里面包含常用的**函数接口**。这是Lambda表达式的基础，Java集合框架也新增部分接口，以便与Lambda表达式对接。

首先回顾一下Java集合框架的接口继承结构：



上图中绿色标注的接口类，表示在Java8中加入了新的接口方法。当然由于继承关系，他们相应的子类也都会继承这些新方法。下表详细列举了这些方法。

接口名	Java8中加入的方法
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使用到 `java.util.function` 包下的接口，这意味着这些方法大部分都跟Lambda表达式相关。我们将逐一学习这些方法。

Collection中的新方法

如上所示，接口 `Collection` 和 `List` 新加入了一些方法，我们以最 `List` 的子类 `ArrayList` 为例来说明，了解Java7 `ArrayList` 实现原理，将有助于理解下文。

forEach()

该方法的签名为 `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，其中 `Consumer` 是个函数接口，里面只有一个待实现方法 `void accept(T t)` （回忆我们自查到，这个方法叫什么根本不重要，你甚至不需要记忆它的名字），

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。

Java7及以前我们可以用增强的for循环实现：

```
// 使用增强for循环遍历
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
for(String str : list){
    if(str.length()>3)
        System.out.println(str);
}
```

现在使用 `forEach()` 方法结合匿名内部类，可以这样实现：

```
// 使用forEach()结合匿名内部类实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.forEach(new Consumer<String>(){
    @Override
    public void accept(String str){
        if(str.length()>3)
            System.out.println(str);
    }
});
```

上述代码调用 `forEach()` 方法，并使用匿名内部类实现 `Consumer` 接口。到目前为止我们没看到这种设计有什么好处，但是不要忘记Lambda表达式。使用Lambda表达式实现如下：

```
// 使用forEach()结合lambda表达式改代
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.forEach( str -> {
        if(str.length()>3)
            System.out.println(str);
    });
```

上述代码给 `forEach()` 方法传入一个Lambda表达式，我们不需要知道 `accept()` 方法，也不需要知道 `Consumer` 接口，类型推导帮我们做了一切。

removeIf()

该方法接收为 `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，里面只有一个特实现方法 `boolean test(T t)` ，同样的这个方法的名字根本不重要，因为用的时候不需要手写这个名字。

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。

我们知道如果需要在迭代过程中对容器进行删除操作会涉及增删元素，否则会抛出 `ConcurrentModificationException` ，所以上述任务传统的写法是：

```
// 使用迭代器删除所有长度大于3的字符串
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
Iterator<String> it = list.iterator();
while(it.hasNext()){
  if(it.next().length()>3) // 删除长度大于3的元素
      it.remove();
}
```

现在使用 `removeIf()` 方法结合匿名内部类，我们可是这样实现：

```
// 使用removeIf()结合匿名类内部类实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.removeIf(new Predicate<String>(){ // 删除长度大于3的元素
  @Override
  public boolean test(String str){
      return str.length()>3;
  }
});
```

上述代码使用 `removeIf()` 方法，并使用匿名内部类实现 `Preicicate` 接口，相信你已经想到用Lambda表达式该怎么写了：

```
// 使用removeIf()结合Lambda表达式实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.removeIf(str -> str.length()>3); // 删除长度大于3的元素
```

使用Lambda表达式不需要记忆 `Predicate` 接口名，也不需要记忆 `test()` 方法名，只需要知道此处需要一个返回布尔类型的Lambda表达式就行了。

replaceAll()

该方法接收为 `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，里面只有一个特实现函数了 `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。

Java7及之前版本没有优雅的办法：

```
// 使用下标实现元素替换
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
for(int i=0; i<list.size(); i++){
  String str = list.get(i);
  if(str.length()>3)
      list.set(i, str.toUpperCase());
}
```

使用 `replaceAll()` 方法结合匿名内部类可以实现如下：

```
// 使用匿名内部类实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.replaceAll(new UnaryOperator<String>(){
  @Override
  public String apply(String str){
      if(str.length()>3)
          return str.toUpperCase();
      return str;
  }
});
```

上述代码调用 `replaceAll()` 方法，并使用匿名内部类实现 `UnaryOperator` 接口，我们知道可以用更为简洁的Lambda表达式实现：

```
// 使用Lambda表达式实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.replaceAll(str -> {
  if(str.length()>3)
      return str.toUpperCase();
  return str;
});
```

sort()

该方法定义在 `List` 接口中，方法签名为 `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，其唯一的方法 `int compare(T o1, T o2)` 需要实现，显然该接口是一个函数接口。

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。

由于Java7以及之前 `sort()` 方法在 `Collections` 工具类中，所以代码要这样写：

```
// Collections.sort()方法
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
Collections.sort(list, new Comparator<String>(){
  @Override
  public int compare(String str1, String str2){
      return str1.length() - str2.length();
  }
});
```

现在可以直接使用 `list.sort()方法` ，结合Lambda表达式，可以这样写：

```
// list.sort()方法结合Lambda表达式
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.sort((str1, str2) -> str1.length() - str2.length());
```

spliterator()

方法签名为 `Spliterator<E> spliterator()` ，该方法返回容器的**可拆分迭代器**，从名字来看该方法跟 `iterator()` 方法有点像，我们知道 `Iterator` 是用来迭代容器的，`Spliterator` 也有类似作用，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` 是可拆分的，一个 `Spliterator` 可以通过调用 `Spliterator<E> trySplit()` 方法来尝试分成两个。一个是 `this` ，另一个是新返回的那个，这两个迭代器代表的元素没有重叠。

可通过（多次）调用 `Spliterator.trySplit()` 方法来分解负载，以便多线程处理。

stream()和parallelStream()

`stream()` 和 `parallelStream()` 分别**返回该容器的** `Stream` **视图表示**，不同之处在于 `parallelStream()` 返回并行的 `Stream` ，`Stream` **是Java函数式编程的核心类**，我们会在后面章节中学习。

Map中的新方法

相比 `Collection` ，`Map` 中加入了更多的方法，我们以 `HashMap` 为例来逐一探秘，了解Java7 `HashMap` 实现原理，将有助于理解下文。

forEach()

该方法接收为 `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，里面有一个特实现方法 `void accept(T t, U u)` ，`BiConsumer` 接口名字和 `accept()` 方法名字都不重要，请不要记忆细节。

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。

Java7以及之前版本的代码如下：

```
// Java7以及之前版本的写法
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
for(Map.Entry<Integer, String> entry : map.entrySet()){
    System.out.println(entry.getKey() + "=" + entry.getValue());
}
```

使用 `Map.forEach()` 方法，结合匿名内部类，代码如下：

```
// 使用forEach()方法结合匿名内部类实现Map
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.forEach(new BiConsumer<Integer, String>(){
    @Override
    public void accept(Integer k, String v){
        System.out.println(k + "=" + v);
    }
});
```

上述代码调用 `forEach()` 方法，并使用匿名内部类实现 `BiConsumer` 接口。当然，实际场景中没人使用匿名内部类写法，因为有Lambda表达式：

```
// 使用forEach()方法结合Lambda表达式实现Map
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.forEach((k, v) -> System.out.println(k + "=" + v));
```

getOrDefault()

该方法跟Lambda表达式没关系，但是很有用。方法签名为 `V getOrDefault(Object key, V defaultValue)` ，作用是**按照给定的** `key` **查询** `Map` **中对应的** `value` **，如果没有找到则返回** `defaultValue` ，使用该方法程序员可以省去查询指定键值是否存在的麻烦。

需求：假设有一个数字到对应英文单词的Map，键35不存在的英文单词，如果不存在则输出Value

```
// 使用Map中的方法，不存在时使用默认值
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
// Java7及之前版本
if(map.containsKey(4)){ // 3
    System.out.println(map.get(4));
}else{
    System.out.println("NoValue");
}
// Java8使用Map.getOrDefault()
System.out.println(map.getOrDefault(4, "NoValue")); // 2
```

putIfAbsent()

该方法跟Lambda表达式没关系，但是很有用。方法签名为 `V putIfAbsent(K key, V value)` ，作用是只有在**不存在** `key` **值的映射或映射值为** `null` **时**，才将 `value` 指定的值放入到 `Map` 中，否则不对 `Map` 做更改，该方法将条件判断和赋值合二为一，使用起来更加方便。

remove()

`remove()` 方法，作用是删除指定 `key` 值的映射，`Map` 中的映射关系：Java8新增了 `remove(Object key, Object value)` 方法，只有在当前 `Map` 中 `key` **正好映射到** `value` 时才删除该映射，否则什么也不做。

replace()

在Java7及以前，要想替换 `Map` 中的映射关系可调用 `put(K key, V value)` 方法实现，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在 `Map` 中加入了两个 `replace()` 方法，分别如下：

- `replace(K key, V value)` ，只有在当前 `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。
- `replace(K key, V oldValue, V newValue)` ，只有在当前 `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。

replaceAll()

该方法接收为 `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，里面有一个特实现方法 `R apply(T t, U u)` 。不要被如此多的函数接口吓到，因为使用的时候根本不需要知道他们的名字。

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。

Java7以及之前版本的代码如下：

```
// Java7以及之前版本的写法-将Map中的映射值大写
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
for(Map.Entry<Integer, String> entry : map.entrySet()){
    entry.setValue(entry.getValue().toUpperCase());
}
```

使用 `replaceAll()` 方法结合匿名内部类，实现如下：

```
// 使用replaceAll()方法结合匿名内部类实现
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.replaceAll(new BiFunction<Integer, String, String>(){
    @Override
    public String apply(Integer k, String v){
        return v.toUpperCase();
    }
});
```

上述代码调用 `replaceAll()` 方法，并使用匿名内部类实现 `BiFunction` 接口，更进一步的，使用Lambda表达式实现如下：

```
// 使用replaceAll()方法结合Lambda表达式实现
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.replaceAll((k, v) -> v.toUpperCase());
```

简洁到让人难以置信。

merge()

该方法接收为 `merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)` ，作用是：

1. 如果 `Map` 中 `key` 对应的映射不存在或者为 `null` ，则将 `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。

参数中 `BiFunction` 函数接口前面已经介绍过，里面有一个特实现方法 `R apply(T t, U u)` 。

`merge()` 方法虽然语文有些复杂，但该方法的用方式很明确，一个比较常见的场景是将新的键值信息拼接到原来的信息上，比如：

```
map.merge(key, newMsg, (v1, v2) -> v1.v2);
```

compute()

该方法接收为 `compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` ，作用是把 `remappingFunction` 的计算结果关联到 `key` 上，如果计算结果为 `null` ，则把 `Map` 中删除 `key` 的映射。

要实现上述 `merge()` 方法中错误信息拼接的例子，使用 `compute()` 代码如下：

```
map.compute(key, (k,v) -> v==null ? newMsg : v.concat(newMsg));
```

computeIfAbsent()

该方法接收为 `V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` ，作用是：只有在当前 `Map` 中不存在 `key` **值的映射或映射值为** `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，里面有一个特实现方法 `R apply(T t)` 。

`computeIfAbsent()` 常用来对 `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，可通过如下代码实现：

```
Map<Integer, Set<String>> map = new HashMap<>();
// Java7及以前的实现方式
if(map.containsKey(1)){
    map.get(1).add("one");
}else{
    Set<String> valueSet = new HashSet<String>();
    valueSet.add("one");
    map.put(1, valueSet);
}
// Java8的实现方式
map.computeIfAbsent(1, v -> new HashSet<String>().add("yi"));
```

使用 `computeIfAbsent()` 将条件判断和添加操作合二为一，使代码更加简洁。

computeIfPresent()

该方法接收为 `V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` ，作用是 `computeIfAbsent()` 相反，即：只有在当前 `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。

这个函数的功能跟如下代码是等效的：

```
// Java7及以前版Composite模式Present工厂函数的代码
if (map.get(key) != null) {
    V oldValue = map.get(key);
    V newValue = remappingFunction.apply(key, oldValue);
    if (newValue != null) {
        map.put(key, newValue);
    }
    else {
        map.remove(key);
    }
    return newValue;
}
return null;
```

1. Java8为容器新增一些有用的方法，这些方法有些是为**竞争条件规避**，有些是为引入**函数式编程**。学习和使用这些方法有助于我们写出更加简洁有效的代码。
2. **函数接口**是前些版本，但绝大多数时候我们根本不需要知道它们的名字，书写Lambda表达式时类型推断帮我们做了一切。

Streams API(I)

你可能没意识到Java对函数式编程的重视程度，看看Java 8加入函数式编程扩充多少功能就清楚了，Java 8之所以费这么大功夫引入函数式编程，原因有二：

1. **代码简洁函数式编程写出的代码语义更图明确，使用stream接口让你从此告别for循环。**
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。

这一节我们学习stream，也就是Java函数式编程的主角，对于Java 7来说stream完全是个陌生东西，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：

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，这些方法将不重载只有，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中间操作调用 `Collection.stream()` 方法得到的，但streamOf.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，一旦遍历过最终数据源之后，就像容器的迭代器那样，想要再次遍历必须重新生成。

对stream的操作分为两类，**中间操作(intermediate operations)和结果操作(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stream，仅此而已。
2. **结果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就会央视。

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：

操作类型	接口方法
中间操作	concat() distinct() filter() flatMap() limit() map() peek() skip() sorted() parallel() sequential() unordered()
结果操作	allMatch() anyMatch() collect() count() findAny() findFirst() forEach() forEachOrdered() max() min() noneMatch() reduce() toArray()

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

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。

forEach()

我们对 `forEach()` 方法熟不陌生，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。

```
// 使用forEach.forEach()遍历
Stream<String> stream = Stream.of("I", "love", "you", "too");
stream.forEach(str -> System.out.println(str));
```

由于 `forEach()` 是结果方法，上述代码会立即执行，输出所有字符串。

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` 条件元素的 `Stream` 。

```
// 使用filter()过滤字符串
Stream<String> stream = Stream.of("I", "love", "you", "too");
stream.filter(str -> str.length()==3)
    .forEach(str -> System.out.println(str));
```

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，如果只调用 `filter()` 不会有实际计算，因此也不会输出任何信息。

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` 。

```
Stream<String> stream = Stream.of("I", "love", "you", "too", "too");
stream.distinct()
    .forEach(str -> System.out.println(str));
```

上述代码会输出去掉一个 `too` 之后的其余字符串。

sorted()

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)` 。

```
Stream<String> stream = Stream.of("I", "love", "you", "too");
stream.sorted(str1, str2 -> str1.length() - str2.length())
    .forEach(str -> System.out.println(str));
```

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。

map()

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。

```
Stream<String> stream = Stream.of("I", "love", "you", "too");
stream.map(str -> str.toUpperCase())
    .forEach(str -> System.out.println(str));
```

上述代码将输出原字符串的大写形式。

flatMap()

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()` 的作用就相当于把由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。

```
Stream<List-Integer> stream = Stream.of(Arrays.asList(1,2), Arrays.asList(3, 4, 5));
stream.flatMap(list -> list.stream())
    .forEach(i -> System.out.println(i));
```

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。

截止到目前我们都觉得挺好，已经有 `Stream` 接口函数理解起来并不费劲儿。如果你就此以为函数式编程不过如此，恐怕是高兴地太早了，下一节对 `Stream` 的操作的介绍将剧新你现有的认识。

Streams API(II)

上一节介绍了部分Stream常见接口方法，理解起来并不困难，但Stream的用法不止于此，本节我们将仍然以Stream为例，介绍流的相关操作。

我们操作（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，都属于我们操作。Stream类库有两个通用的我们操作 `reduce()` 和 `collect()` ，也有一些为简化书写而设计的专用我们操作，比如 `sum()` 、 `max()` 、 `min()` 、 `count()` 等。

最大或最小值这类我们操作很好理解（至少方法语义上是这样），我们看看介绍 `reduce()` 和 `collect()` ，这是比较有魔法的地方。

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()` 最常用的结果就是从一堆值中生成一个值，用这么复杂的函数去表示一个最大或最小值，你都不觉得很设计者有病，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”。

```
// 初始最长的单词
Stream<String> stream = Stream.of("I", "love", "you", "too");
Optional<String> longest = stream.reduce((s1, s2) -> s1.length()>=s2.length() ? s1 : s2);
//Optional<String> longest = stream.max((s1, s2) -> s1.length()-s2.length());
System.out.println(longest.get());
```

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：*求出一部单词的长度之和*，这是个求和”操作，操作对象输入类型是String，而结果类型是Integer。

```
// 初始字符串之和
Stream<String> stream = Stream.of("I", "love", "you", "too");
Integer lengthSum = stream.reduce(0, // 初始值 // (1)
        (sum, str) -> sum.str.length(), // 累加器 // (2)
        (a, b) -> a+b); // 部分结果组合，并行执行才会用到 // (3)
// int lengthSum = stream.mapToInt(str -> str.length()).sum();
System.out.println(lengthSum);
```

上述代码稍有边绕，字符串被射成长度，a, b并和当前累加和相加，这显然是再步操作，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()` 提供求总量上述目的，也是可以的。

`reduce()` 理论的是生成一个值，如果想要从Stream生成一个集合或者Map等复杂的时候该怎么办呢？终极武器 `collect()` 横空出世！

终极武器collect()

不夸张的讲，如果你发现某个功能在Stream接口中找找到，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，先看个简单的小例子：

```
// 把字符串转换为大写字符串
Stream<String> stream = Stream.of("I", "love", "you", "too");
List<String> list = stream.collect(Collectors.toList());// (1)
// Set<String> set = stream.collect(Collectors.toSet());// (2)
// HashSet<String, Integer> map = stream.collect(Collectors.toMap(Function.identity(), String::length));// (3)
```

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：

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

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？这要从再方圆解：

1. 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，`Function.identity()` 就是*Function*接口的一个静态方法。
2. `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方法来避免写了的工具类呢？

方法引用

请知 `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：

方法引用类型	举例
引用静态方法	`Integer::sum`
引用某个对象的方法	`list::add`
引用某个类的方法	`String::length`
引用构造方法	`HashMap::new`

我们会在后面的例子中使用方法引用。

收集器

相信前面繁琐的内容已经磨灭了你学习Java函数式编程的热情，不过很遗憾，下面的内容更繁琐。但这不能怪Stream类库，因为要实现的功能本身就复杂。

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：

1. 目标容器是什么？是ArrayList还是HashSet，或者是TreeMap。
2. 新元素如何添加到容器中？是List.add()还是Map.put()。

如果并行的进行呢，还需要告诉*collect()* 3. 多个部分结果如何合并成一个。

理解了这些，*collect()*方法有个这样的重载形式：`<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个要素，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*工具类可谓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：

```
// 将字符串规约成List
Stream<String> stream = Stream.of("I", "love", "you", "too");
List<String> list = stream.collect(Collectors.toList());// (1)
Set<String> set = stream.collect(Collectors.toSet());// (2)
```

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。

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，这里再瓦一下，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：

```
// 将Stream转换成List或Set
Stream<String> stream = Stream.of("I", "love", "you", "too");
List<String> list = stream.collect(Collectors.toList());// (1)
Set<String> set = stream.collect(Collectors.toSet());// (2)
```

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现类型，这个需求可通过 `Collectors.toCollection(Supplier<C> collectionFactory)` 方法完成：

```
// 指定HashSet和ArrayList容器的具体类型
ArrayList<String> arrayList = stream.collect(Collectors.toCollection(ArrayList::new));// (3)
HashSet<String> hashSet = stream.collect(Collectors.toCollection(HashSet::new));// (4)
```

上述代码功能指定我的结果是*ArrayList*，而*(4)*处指定我的结果为*HashSet*，一切如你所愿。

使用collect()生成Map

前面已经说过Stream有些类似于某种数据源，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，概本质是数据流量是增是减，通常在上，通常在上，种情况下 collect() 的结果是Map!

1. 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。
2. 使用 Collectors.partitioningBy() 生成的收集器，对元素进行二分区操作时用到。
3. 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，这里和 Collectors.toCollection() 并列的方法。如下代码展示所学生列表转换成id-学生，GPA组成的Map，非常直观，无需多言。

```
// 假设Mapped是以学生学号为key的Map<Student, Double> StudentToGPA = students.stream().collect(Collectors.toMap(Function.identity(),// 如何生成key                                                                student -> computeGPA(student)));// 如何生成value
```

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。下例代码展示将学生分成成绩及格或不及格的两部分。

```
// Partition students into passing and failing Map<Boolean, List<Student>> passingFailing = students.stream()        .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

情况3：使用 groupingBy() 生成的收集器，这是比较灵活的一种情况，更多地用在groupingBy语句实现，这里的groupingBy的意思是按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。下例代码展示将员工按照部门进行分组：

```
// Group employees by department Map<Department, List<Employee>> byDept = employees.stream()        .collect(Collectors.groupingBy(Employee::getDepartment));
```

以上只是分组的最基本用法，有些时候仅分组是不够的，在SQL中使用group by是为了协助其他查询，比如*统计员工规模的门分组，2. 然后统计每个部门员工的人数*，Java的高级语法中也提供了这种功能，增强版的 groupingBy() 接收第三个参数，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，这种为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)。

```
// 使用下游收集器统计每个部门的人数 Map<Department, Integer> totalByDept = employees.stream()        .collect(Collectors.groupingBy(Employee::getDepartment,                                       Collectors.counting()));// 下游收集器
```

上面代码的进阶是不是越看越像SQL？高度非结构化，可视效果极佳。下游收集器还可以往往变更下游的收集器，这感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结果，**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：

```
// 按照部门对员工分类，并只保留员工的名字 Map<Department, List<String>> byDept = employees.stream()        .collect(Collectors.groupingBy(Employee::getDepartment,                       Collectors.mapping(Employee::getName,// 下游收集器                                          Collectors.toList())));// 第二下游收集器
```

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。

使用collect()做字符串join

这个用法是大家最熟悉的功能，字符串拼接时使用 Collectors.joining() 生成的收集器，从此告别Oh shit！ 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，代码证自难念。

```
// 使用Collectors.joining()拼接字符串String String1 = Stream.of("I", "love", "you");//String joined = stream.collect(Collectors.joining());// "Ilovyou"//String joined = stream.collect(Collectors.joining(","));// "I,love,you"String joined = stream.collect(Collectors.joining(",", "{", "}"));// "{I,love,you}"
```

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结果**，不过Collectors工具类该装能满足我们的绝大部分需求，手动实现之间流先看看文档。

Stream Pipelines

前面我们已经学会如何使用Stream API，用起来真的很爽，但是别的方法下面似乎隐藏着无尽的秘密，如此强大的API是如何实现的呢？比如Pipeline是怎么执行的，每次方法调用都会导致一次迭代吗？自由时代又是怎么做到的，线程个数是多少？本节我们学习Stream流水线的原理，这是Stream实现的关键所在。

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：

```
// Arrayylist.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    }    ...}
```

我们看到 Arraylist.forEach() 方法实现的逻辑就是一个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，但这样不是关键，理解Stream我们要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：

```
int longestStringLength=StartWithN(Stream    .stringStream()            .filter("a -> a.startsWith("A");            .map(String::length)            .max();
```

上述代码表达以字母A开头的字符串的最大长度。一种直自的方式是为每一次函数调用新增一次迭代，这样做能够实现功能，但效率上肯定是无法接受的，类库的实现看使用流水线（Pipeline）的方式巧妙的避了多次迭代，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操作分类	
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()
结果操作(Terminal operations)	非短路操作	forEach() forEachOrdered() toArray() reduce() collect() max() min() count()
	短路操作(short-circuiting)	anyMatch() allMatch() noneMatch() findFirst() findAny()

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；结果操作又可以分为短路操作和非短路操作，短路操作是指不需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，是因为底层并行执行的关键就在于此。

一种直白的实现方式

仍然考虑上述求长字符串的程序，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() 方法后应该时，流水线中间结果会以某种容器的形式存下来，比如filter方法应该时，生成的结果放到List中，接着调用 map方法时，生成的结果放到Map中，最后遍历list找出最大的数字作为最终结果。程序的执行流程如图所示：

这种做实现起来非常简单直观，但有两个明显的弊端：

1. 迭代次数多，迭代次数跟函数调用的次数相等。
2. 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。

这些弊端使效率低下，根本无法接受，如果不使用Stream API我们都知道上述代码说轮何在一次迭代中完成，大致如下形式：

```
int longest = 0;for(String str : strings){    if(str.startsWith("A")){// 1. Filter()，保留以A开头的字符串        int len = str.length();// 2. mapToInt()，转换成长度        longest = Math.max(len, longest);// 3. max()，保留最长的长度    }}
```

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，只需我们事先知道用户意图，总是能够早用上迭代方式实现跟Stream API等价的功能，但问题是Stream要面对的是开发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。

Stream流水线解决方案

我们大致能够感觉，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才能记录的操作叠加到一起在一次迭代中全部执行掉，治着这个思路，有几个问题需要解决：

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

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*方法*，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*一个数据来源，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，并用某种类似后的Pipeline辅助来代表Stage，具体有先后顺序的各个Stage首尾一起，就构成了整个流水线。接Stream相关类和接口的继承关系图示：

还有四个Pipeline, *AbstractPipeline*, *DoublePipeline*没在图中画出，这三个类引为三种基本类型（不是包装类型）而定制的，除了ReferencePipeline是并列关系，图中Head用于表示第一个Stage，即调用静态方法Collection.stream()方法产生的Stage，很显然这个Stage是不包含任何操作：*StatelessOpStatefulOp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流水线相关结构示意图如下：

图中清汪 Collection.stream() 方法调到Head方法就是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，结果这种结构就能够以时期重源的形有操作**，这就是Stream记录操作的方式。

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，这样起来似乎是可行的，但是你忘了前面的问题*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和本身才知道该如何执行自己但自己的动作，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：

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，并对元素进行处理。Stage把自己包含的操作和回调方法封装到该方法里，前一个Stage只需要调用当前Stage.accept(T t)方法就行了。

有了上面的协议，相邻Stage之间调用就很容易了，每个Stage都会调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的实现 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()方法可能创建一个调数组缓存的容器，而accept()方法则是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，只要收到一个元素，cancellationRequested就应该返回true，以便调用者尽快结束管道，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内部实现线程的本质，就是如何调度和Sink的这四个接口方法。**

有了Sink序列的保证，Stage之间的调用问题就解决了，执行时只需要从流水线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：

```
void accept(U u){    1. 使用当前Sink保存的回调函数处理u    2. 将处理结果传递给流水线下游的Sink}
```

Sink接口的其他几个方法也是按照这种比葫一转发的模型实现，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的方法：

```
// Stream.map()，调用流水线的第一个阶段的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) {        @Override        /*ReferencePipeline子类匿名内部类重写抽象函数返回Sink*/        Sink<P_OUT> opWrapSink(int flags, Sink<R. downstream) {            return new Sink.ChainedReference<P_OUT, R>(downstream) {                @Override                public void accept(P_OUT w) {                    R r = mapper.apply(w);// 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                }            };        }    };}
```

上述代码看似复杂，其实逻辑简单明瞭，就是将回调函数mapper包装到一个Sink当中。由于Stream.map必是一个无状态的中间操作，所以map方法返回了一个StatelessOp的部类对象（一个新的Stream），调用这个新Stream的opWrapSink()方法得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()方法调用时Stream中的元素进行排序，显然这是一个有状态的中间操作，因为该取所有元素之前是没法得到最终顺序的，跳并概念代码直接进入同基本类，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：

```
// Stream.sort()方法调用时Sink实现class RefSortingSink<T> extends AbstractRefSortingSink<T> {    private Arrayylist<T. list;// 容纳排序结果的元素    RefSortingSink(Sink<T> sink? Sort<T> downstream, Comparator<? super T> comparator) {        super(downstream, comparator);    }    @Override    public void begin(long size) {        ...        // 初始一个容纳结果元素的容器        list = (size >= 0) ? new Arrayylist-T-((int) size) : new Arrayylist-T-();    }    @Override    public void end() {        list.sort(comparator);// 只有元素全部聚合之后才能开始排序        downstream.begin(list.size());        if (!cancellationRequested(*//* 下游Sink不在短路操作        list.forEach(downstream::accept);// 2. 将处理结果传递给流水线下游的Sink        ...    }    ...    else {// 下游Sink有短路操作        for (T t : list) {// 每次调用Sink.cancellationRequested()询问是否可以结束了。            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;    ...    @Override    public void accept(T t) {        list.add(t);// 1. 使用当前Sink处理u的方法：将元素存入容器中缓存，只是简单的将元素添加到中间容器当中    }}
```

上述代码实现的调用了Sink的四个接口方法是如何协同工作的：

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一步操作，并提出了三个接口方法，**使用这些方法，流水线就能以一种高效的方式执行**。结果操作会创建一个包装了自己操作的Sink，这些是最终结果中最后一个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的处理一样发现蛋，结果操作的Sink就是调用链的尾巴。

我们再来看看一个复杂的例子，Stream.sorted()方法调用时Stream中的元素进行排序，显然这是一个有状态的中间操作，因为该取所有元素之前是没法得到最终顺序的，跳并概念代码直接进入同基本类，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：

```
// Stream.sort()方法调用时Sink实现class RefSortingSink<T> extends AbstractRefSortingSink-T- {    private Arrayylist-T- list;// 容纳排序结果的元素    RefSortingSink(Sink<T> sink? Sort<T> downstream, Comparator<? super T> comparator) {        super(downstream, comparator);    }    @Override    public void begin(long size) {        ...        // 初始一个容纳结果元素的容器        list = (size >= 0) ? new Arrayylist-T-((int) size) : new Arrayylist-T-();    }    @Override    public void end() {        list.sort(comparator);// 只有元素全部聚合之后才能开始排序        downstream.begin(list.size());        if (!cancellationRequested*()*) {// 下游Sink不在短路操作        list.forEach(downstream::accept);// 2. 将处理结果传递给流水线下游的Sink        ...    }    ...    else {// 下游Sink有短路操作        for (T t : list) {// 每次调用Sink.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;    ...    @Override    public void accept(T t) {        list.add(t);// 1. 使用当前Sink处理u的方法：将元素存入容器中缓存，只是简单的将元素添加到中间容器当中    }}
```

上述代码完美的调用了Sink的四个接口方法是如何协同工作的：

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。

Sink完美封装了Stream一步操作，并提出了三个接口方法，**使用这些方法，流水线就能以一种高效的方式执行**。结果操作会创建一个包装了自己操作的Sink，这些是最终结果中最后一个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的处理一样发现蛋，结果操作的Sink就是调用链的尾巴。

我们再来看看一个复杂的例子，Stream.sorted()方法调用时Stream中的元素进行排序，显然这是一个有状态的中间操作，因为该取所有元素之前是没法得到最终顺序的，跳并概念代码直接进入同基本类，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：

```
// AbstractPipeline.wrapSink()// 从下游的上游不断创建Sink，如果遇到中间操作则创建中间操作// 创建返回新的管道时调用一次来创建上游和操作的Sink。final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {    ...    for (AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p=p.previousStage) {        ...        // 调用当前Stage的opWrapSink()方法将当前Stage的操作封装成Sink        Sink ns = p.opWrapSink(p.1.1flags, sink);        ...        // 将当前Stage的Sink赋值给下一个Stage        sink = ns;    }    ...    return (Sink<P_IN>) sink;    ...}// ReferencePipeline.opWrapSink()// 中间操作子类实现opWrapSink()方法，将回调函数包装成Sink返回@Override abstract Sink<U> opWrapSink(int flags, Sink<P_OUT> sink);// ReferencePipeline.StatelessOp.opWrapSink()// 无状态中间操作子类实现opWrapSink()方法，将回调函数包装成Sink返回@Override public final Sink<P_IN> opWrapSink(int1 flags, Sink<P_OUT> sink) {    assert getOutputShape() == getInputShape();    int1 lff = getOpFlags();    int1 lsf = getStrlfOpFlag();    int1 flags = StreamOpFlag.1.1compose(lff, flags);    boolean1 isStateful = lsf == STATEFUL_OP;    Sink<P_OUT> s = sink;    for (AbstractPipeline1 1.1p=this.1.1previousStage; 1.1p.1.1depth > 0; 1.1p=1.1p.1.1previousStage, isStateful = false) {        int1 lsf = 1.1p.1.1getStrlfOpFlag();        int1 lff = 1.1p.1.1getOpFlags();        flags = StreamOpFlag.1.1compose(lff, flags);        boolean1 1.1isStateful = lsf == STATEFUL_OP;        s = new ChainedReference<>(1.1p.1.1sourceStage, 1.1p.1.1upstream, s);        1.1p.1.1sourceStage.1.1sinkFlags = flags;        1.1p.1.1upstream = (Sink<Object>) s;        1.1p.1.1stateful = 1.1isStateful;    }    return (Sink<P_IN>) s;}// ReferencePipeline.StatefulOp.opWrapSink()// 有状态中间操作子类实现opWrapSink()方法，将回调函数包装成Sink返回@Override abstract Sink<U> opWrapSink(int1 1.1flags, Sink<P_OUT> 1.1sink);// DistinctOp.opWrapSink()// 有状态中间操作distinct()实现opWrapSink()方法，将回调函数包装成Sink返回@Override public final Sink<T> opWrapSink(int1 1.1flags, Sink<T> 1.1sink) {    assert getOutputShape() == getInputShape();    int1 1.1lff = getOpFlags();    int1 1.1lsf = getStrlfOpFlag();    int1 1.1flags = StreamOpFlag.1.1compose(1.1lff, 1.1flags);    boolean1 1.1isStateful = 1.1lsf == STATEFUL_OP;    Sink<T> 1.1s = 1.1sink;    if (1.1isStateful && StreamOpFlag.DISTINCT.1.1isKnown(1.1flags)) {        return 1.1s;    }    else {        if (1.1isStateful && StreamOpFlag.ORDERED.1.1isKnown(1.1flags)) {            return (Sink<T>) new DistinctOrdered<>(1.1s);        }        else {            return (Sink<T>) new DistinctUnordered<>(1.1s);        }    }}
```

上述代码实现的调用了Sink的四个接口方法是如何协同工作的：

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。

并行流

Stream API的一大亮点是可以轻松的将流水线的执行方式从单线程模式变成多线程模式，这一功能的实现是在Pipeline辅助中设置一个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) 方法来调用Sink，该方法的作用是返回一个新的包含了当前Stage代表的操作及接收调用结果传递给downstream的Sink对象，为什么要产生一个新对象而不是直接一个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）组合成新Sink，这就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()// 从下游的上游不断创建Sink，如果遇到中间操作则创建中间操作// 创建返回新的管道时调用一次来创建上游和操作的Sink。final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {    ...    for (AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p.previousStage) {        ...        // 调用当前Stage的opWrapSink()方法将当前Stage的操作封装成Sink        Sink ns = p.opWrapSink(p.1.1flags, sink);        ...        // 将当前Stage的Sink赋值给下一个Stage        sink = ns;    }    ...    return (Sink<P_IN>) sink;    ...}
```

上述代码实现的调用了Sink的四个接口方法是如何协同工作的：

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。

并行流的实现方式

Stream API的一大亮点是可以轻松的将流水线的执行方式从单线程模式变成多线程模式，这一功能的实现是在Pipeline辅助中设置一个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) 方法来调用Sink，该方法的作用是返回一个新的包含了当前Stage代表的操作及接收调用结果传递给downstream的Sink对象，为什么要产生一个新对象而不是直接一个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）组合成新Sink，这就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()// 从下游的上游不断创建Sink，如果遇到中间操作则创建中间操作// 创建返回新的管道时调用一次来创建上游和操作的Sink。final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {    ...    for (AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p.previousStage) {        ...        // 调用当前Stage的opWrapSink()方法将当前Stage的操作封装成Sink        Sink ns = p.opWrapSink(p.1.1flags, sink);        ...        // 将当前Stage的Sink赋值给下一个Stage        sink = ns;    }    ...    return (Sink<P_IN>) sink;    ...}
```

上述代码实现的调用了Sink的四个接口方法是如何协同工作的：

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。

并行流的实现方式

Stream API的一大亮点是可以轻松的将流水线的执行方式从单线程模式变成多线程模式，这一功能的实现是在Pipeline辅助中设置一个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) 方法来调用Sink，该方法的作用是返回一个新的包含了当前Stage代表的操作及接收调用结果传递给downstream的Sink对象，为什么要产生一个新对象而不是直接一个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）组合成新Sink，这就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()// 从下游的上游不断创建Sink，如果遇到中间操作则创建中间操作// 创建返回新的管道时调用一次来创建上游和操作的Sink。final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {    ...    for (AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p.previousStage) {        ...        // 调用当前Stage的opWrapSink()方法将当前Stage的操作封装成Sink        Sink ns = p.opWrapSink(p.1.1flags, sink);        ...        // 将当前Stage的Sink赋值给下一个Stage        sink = ns;    }    ...    return (Sink<P_IN>) sink;    ...}
```

上述代码实现的调用了Sink的四个接口方法是如何协同工作的：

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。

```
        sink = p.opWrapSink(p.previousStage.combinedFlags, sink);
    }
    return (Sink<P_IN>) sink;
}
```

现在流水线上从开始到结果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(Sink<P_IN> wrappedSink, Spliterator<P_IN> spliterator) {
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);// 遍历
        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结果（事实上，除了打印之外其他结果都应避免使用副作用），对于真正需要返回结果的结果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()都是归约的操作，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。

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那理并写出正确的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。

注：留下本文所用的JDK版本，以便有考究癖的人考证：

```
$ java -version
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) Server VM (build 25.181-b13, mixed mode)
```

文章来源：https://github.com/CarpenterLee/JavaLambdaInternals

更多精彩文章请关注作者维护的公众号「后端进阶」，这是一个专注后端相关技术的公众号。关注公众号并回复「后端」免费领取后端相关电子书籍。 欢迎分享，转载请保留出处。

微信公众号「JavaR科代表」

上一篇 Docker Overlay网络的一些总结
下一篇 RocketMQ源码分析之路由中心

9 条评论 未登录用户

说点什么

支持 Markdown 语法 使用 GitHub 登录 预览

tslye 发表于 超过 2 年前
优秀

hqfu 发表于 超过 2 年前
学习了

zhengyuefeng 发表于 超过 2 年前
秀

elbertwang 发表于 超过 2 年前
写的挺好啊，多谢~已经学习了

objcoding 发表于 超过 2 年前
@elbertwang
写的挺好啊，多谢~已经学习了
关注我的公众号「后端进阶」，更多干货及时推送😄

lazyTimes 发表于 大约 2 年前
学习了，写得很棒

sherpzhangs 发表于 两正 2 年前
作者说的重载其实都是重写吧

KeepSilenceQP 发表于 两正 2 年前
点赞

ghost 发表于 超过 1 年前
自定义函数接口那里可以写成这样

```
CustomizeLambda<String> customizeLambda = System.out::println;
```