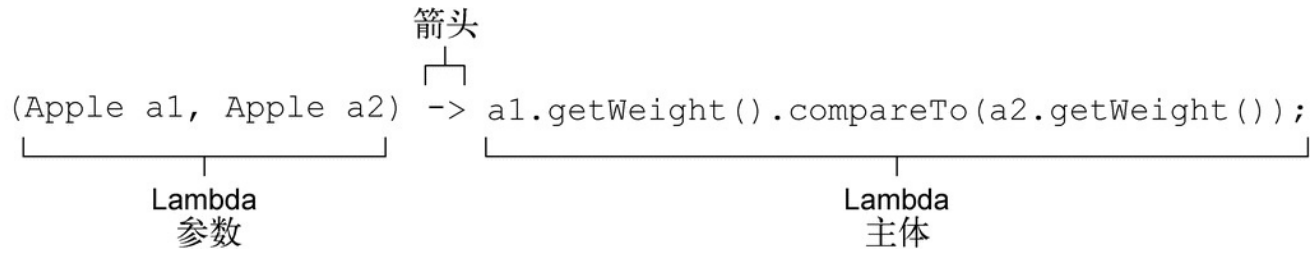
1 Lambda表达式

lambda表达式由参数、箭头和函数主体组成。可以把Lambda表达式理解为简洁地表示可传递的匿名函数的一种方式：它没有名称，但它有参数列表、函数主体、返回类型，可能还有一个可以抛出的异常列表。



- `(Apple a1, Apple a2)`: 参数列表 — 这里它采用了Comparator中compare方法的参数，两个Apple。
- `->`: 箭头， 把参数列表与Lambda主体分开。
- `Lambda主体`: 比较两个Apple的重量。表达式就是Lambda的返回值了。

Lambda的基本语法是

```
(parameters) -> expression
(parameters) -> {statements;} // 注意花括号和分号
```

函数式接口

函数式接口(funtional interface)就是定义且只定义了一个抽象方法的接口。函数式接口的抽象方法的签名称为函数描述符。函数式接口可以带 有`@FunctionalInterface`的注解，但不是必须的。常见的函数式接口有`Comparable`, `Runnable`, `Callable`。

```
// Runnable.java
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

用函数式接口可以干什么呢？Lambda表达式允许你直接以内联的形式为函数式接口的抽象方法提供实现，并把整个表达式作为函数式接口的实例 （具体说来，是函数式接口一个具体实现的实例):

```
// 使用lambda
Runnable r1 = () -> System.out.println("hello world!");
// 使用匿名类
Runnable r2 = new Runnable() {
    public void run() {
        System.out.println("Hello world!");
```

```
    }
};

public static void process(Runnable r) {
    r.run();
}
// 利用直接传递的Lambda
process(()->System.out.println("hello world!"));
```

函数描述符

函数式接口的抽象方法就是函数描述符。例如，`Runnable`接口可以看作一个什么也不接受什么也不返回(void)的函数的签名，因为它只有一个叫作`run()`的抽象方法，这个方法什么也不接受，什么也不返回(void)。可以写成`()->void`来描述函数式接口的签名。

一个例子

资源处理(例如处理文件或数据库)是一个常见的模式：打开一个资源，做一些处理，然后关闭资源。这个设置和清理阶段总是很类似，并且会围绕着执行处理的那些重要代码。这就是所谓的环绕执行(execute around)模式，下图很好的展示了环绕执行模式的特点：

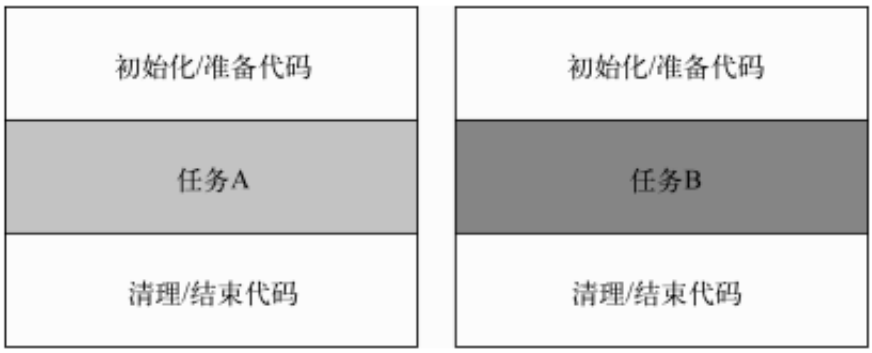


图3-2　任务A和任务B周围都环绕着进行准备/清理的同一段冗余代码

带资源的try语句块，会在结束后释放资源。而核心代码只有 **br.readLine()**

```
public static String processFile() throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

假如我们下次需要读取文件前两行呢？我们可能需要复制一下上面的方法。如下：

```
public static String processFile() throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

如果现在需要读取三行，最后一行呢？会造成太多代码冗余了！但是java8后你可以这样：

1.定义一个函数式接口

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

2.定义读取文件的方法

```
public static String processFile(BufferedReaderProcessor p)
                                      throws IOException {
 try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
    return p.process(br);
 }}
```

3.行为参数化-传递lamdba行为表达式

```
//处理一行:
String oneLine = processFile((BufferedReader br) -> br.readLine());
//处理两行:
String twoLines = processFile((BufferedReader br) -> br.readLine()
            + br.readLine());
```

常见的函数式接口

Java 8在`java.util.function`包中引入了几个新的函数式接口。

表3-2　Java 8中的常用函数式接口

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

Predicate

Predicate接口定义了一个名叫test的抽象方法，它接受泛型T对象，并返回一个boolean。

以下是源码的一部分：

```
@FunctionalInterface
public interface Predicate<T>{
    boolean test(T t);
}

// filter源码部分
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T s: list){
        if(p.test(s)){
            results.add(s);
        }
    }
 return results;
}
```

```
// 实际使用场景-1
Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
// 实际使用场景-2
List<String> nonEmpty = filter(listOfStrings, (String s) -> !s.isEmpty());
```

Consumer

Consumer定义了一个名叫accept的抽象方法，它接受泛型T的对象，没有返回（void）。你如果需要访问类型T的对象，并对其执行某些操作，就可以使用这个接口.

```
@FunctionalInterface
public interface Consumer<T>{
  void accept(T t);
}

public static <T> void forEach(List<T> list, Consumer<T> c){
     for(T i: list){
         c.accept(i);
    }
}

// 实际使用--lambda表达式即为Consumer函数式接口参数
forEach(
    Arrays.asList(1,2,3,4,5),
    (Integer i) -> System.out.println(i)
 );
```

Function

Function<T, R>接口定义了一个叫作apply的方法，它接受一个泛型T的对象，并返回一个泛型R的对象

```
@FunctionalInterface
public interface Function<T, R>{
    R apply(T t);
}
public static <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T s: list){
        result.add(f.apply(s));
    }
 return result;
}

// 这里 (String s)相当于T-String  s.length()相当于 R --int
List<Integer> l = map(
```

```
        Arrays.asList("lambdas","in","action"),
        (String s) -> s.length()
);
```

原始类型特化

我们介绍了三个泛型函数式接口：Predicate、Consumer和Function<T,R>。还有些函数式接口专为某些类型而设计。

> 如果基础类型也使用这些函数式接口，比如Predicate通过自动拆箱装箱是可以实现的，但这在性能方面是要出代价的。装箱的本质就是把原来的原始类型包装起来，并保存在堆里。因此，装箱后值需要更多的内存，并需要额外的内存搜索来获取被包装的原始值。

Java 8为我们前面所说的函数式接口带来了一个专门的版本，以便在输入和输出都是原始类型时避免自动装箱的操作。

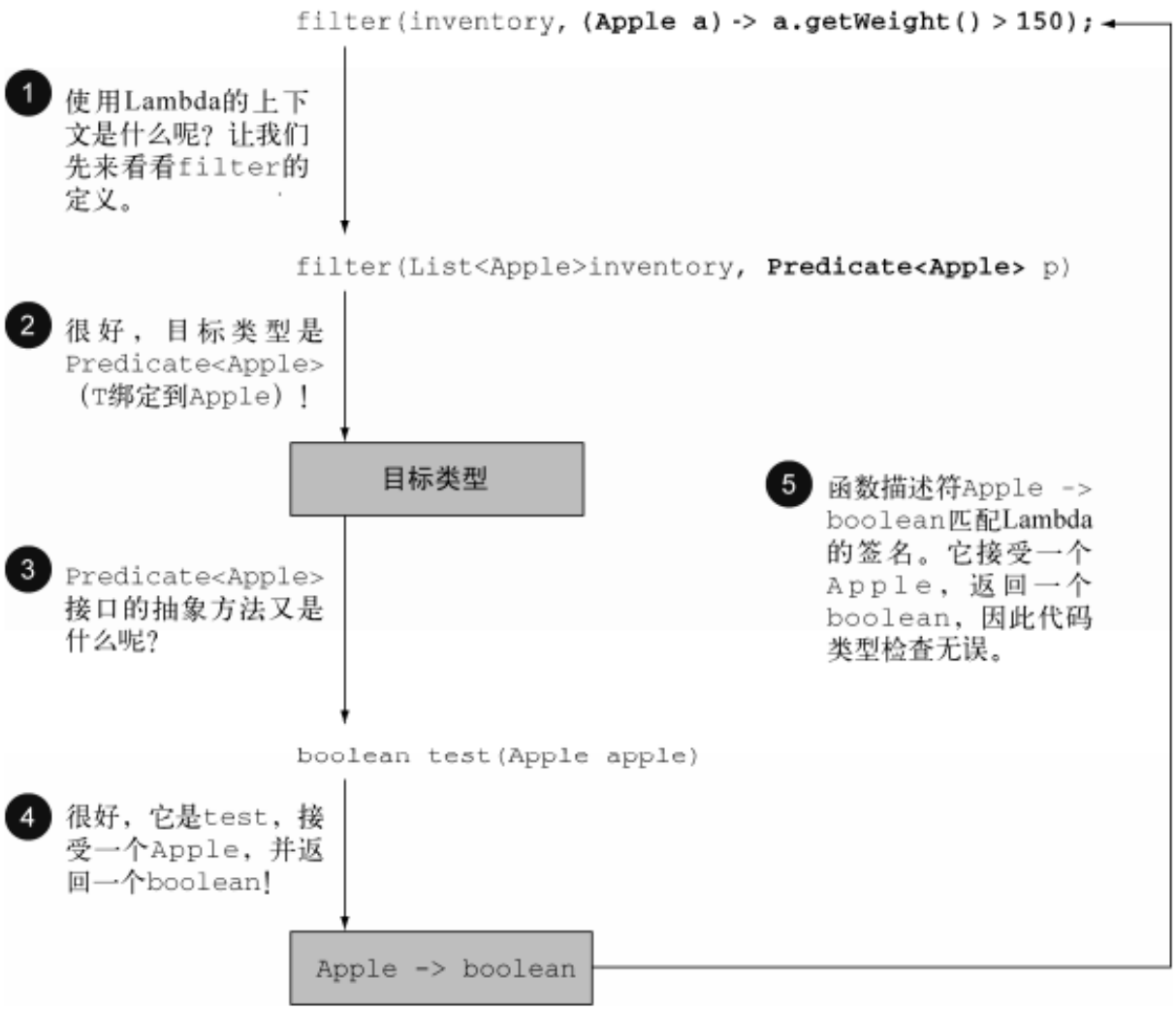
类型检查推断

> 当我们第一次提到Lambda表达式时，说它可以为函数式接口生成一个实例。然而，Lambda表达式本身并不包含它在实现哪个函数式接口的信息。

类型检查

Lambda的类型是从使用Lambda的上下文推断出来的。上下文（比如，接受它传递的方法的参数，或接受它的值的局部变量）中Lambda表达式需要的类型称为目标类型。让我们通过一个例子，看看当你使用Lambda表达式时背后发生了什么。

```
List<Apple> heavierThan150g =
        filter(inventory, (Apple a) -> a.getWeight() > 150);
```



请注意，如果Lambda表达式抛出一个异常，那么抽象方法所声明的throws语句也必须与之匹配

> **特殊的void匹配规则**
>
> 如果一个Lambda的主体是一个表达式，就和一个返回void的函数式接口兼容。（当然参数列表必须兼容）
>
> 例如，以下两行都是合法的，尽管List的add方法返回了一个boolean，而不是函数Consumer上下文（T -> void）所要求的void：

```
//Predicate返回了一个boolean
Predicate<String> p = s -> list.add(s);
//Consumer返回了一个void
Consumer<String> b = s -> list.add(s);
```

类型推断

Java编译器会像下面这样推断Lambda的参数类型：

```
// 参数a没有显示说明类型
List<Apple> greenApples = filter(inventory, a -> "green".equals(a.getColor()));
```

Lambda表达式有多个参数，代码可读性的好处就更为明显。例如，你可以这样来创建一个Comparator对象：

```
// 没有类型推断
Comparator<Apple> c = (Apple a1, Apple a2) ->
a1.getWeight().compareTo(a2.getWeight());
// 有类型推断
Comparator<Apple> c = (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

有时候显式写出类型更易读，有时候去掉它们更易读。没有什么法则说哪种更好；对于如何让代码更易读，程序员必须做出自己的选择。

使用局部变量

我们迄今为止所介绍的所有Lambda表达式都只用到了其主体里面的参数。但Lambda表达式也允许使用自由变量（不是参数，而是在外层作用域中定义的变量），就像匿名类一样。 它们被称作捕获Lambda

下面的Lambda捕获了portNumber变量：

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

尽管如此，还有一点点小麻烦：关于能对这些局部变量做什么有一些限制。Lambda可以没有限制地捕获（也就是在其主体中引用）实例变量和静态变量。但是局部变量必须显式声明为final，或事实上是final。换句话说，Lambda表达式只能捕获局部变量一次。（注：捕获实例变量可以被看作捕获最终局部变量this。） 例如，下面的代码无法编译，因为portNumber变量被赋值两次：

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
// 错误的，因为portNumber被赋值两次，lambda捕获的局部变量必须是显示final或事实上就是final（即只被赋值一次的）
```

> **为什么这样限制**
>
> 实例变量不需要是final，而局部变量需要是final。最主要的原因是因为：实例变量是存储在堆上的，而局部变量是存储在方法栈上的。而当lambda函数访问局部变量时，该变量可能已经被回收，因此只会捕获一次即只会复制一次局部变量的副本，访问时即访问副本。因此该变量必须保证是final，副本才有效。

使用方法引用

方法引用其实就是为了使代码可读性更高，例如：

```
// 直接使用lambda
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
// 使用方法引用
inventory.sort(comparing(Apple::getWeight));
```

方法引用可以被看作仅仅调用特定方法的Lambda的一种快捷写法。它的基本思想是，如果一个Lambda代表的只是“直接调用这个方法”，比如`(Apple a) -> a.getWeight()`，那最好还是方法引用来调用它：`Apple::getWeight`

当你需要使用方法引用时，目标引用放在分隔符::前，方法的名称放在后面。`Apple::getWight` 即Apple是目标引用

```
() -> Thread.currentThread().dumpStack()  ==>
 Thread.currentThread()::dumpStack
(str, i) -> str.substring(i)              ==>    String::substring
```

如何构建方法引用

- 指向静态方法的方法引用

```
(agrs) -> ClassName.staticMethod(args)    ==>    ClassName::staticMethod
```

- 指向任意类型实例方法的引用

```
(exp,args) -> exp.instanceMethod(args)    ==>    ExpClassName::instanceMethod
```

- 指向现有对象的实例方法的方法引用

```
// exp是已有变量
(args) -> exp.instanceMethod(args)        ==>    exp::insatanceMethod
```

> 编译器会进行一种与Lambda表达式类似的类型检查过程，来确定对于给定的函数式接口，这个**方法引用**是否有效：**方法引用的签名必须和上下文类型匹配**。

构造函数引用

- 无参构造函数引用 即 () -> T

```
Supplier<Apple> = Apple:new
```

- 一个参数构造函数引用 即 (P) -> T

```
Function<Integer, Apple> = Apple::new
```

- 两个参数的构造函数引用 即 (P1, P2) -> T

```
BiFunction<Integer, Integer, Apple> = Apple::new
```

- 多个构造函数的引用也一样，只是需要自定义函数式接口。接口上下文符合 (P1,P2,P3...) -> T 即可

一个栗子：

```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
// 调用map方法获得一组apple实例的集合
List<Apple> apples = map(weights, Apple::new);
// 将构造函数引用传递给map方法
public static List<Apple> map(List<Integer> list, Function<Integer, Apple> f){
    List<Apple> result = new ArrayList<>();
    for(Integer e: list){
        result.add(f.apply(e));
    }
    return result;
}
```

不将构造函数实例化却能够引用它，这个功能有一些有趣的应用。比如下面的giveMeFruit方法可以获得各种各样不同重量的水果实例：

```
// 创建一个Map 字符串映射相应的构造函数引用
static Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
    // apple 匹配Apple的构造函数引用
    map.put("apple", Apple::new);
    map.put("orange", Orange::new);
    // etc...
}
// 这个方法可以通过输入的 fruit名字 以及构造函数需要的参数，获得相应的实例。
public static Fruit giveMeFruit(String fruit, Integer weight){
    return map.get(fruit.toLowerCase())
              .apply(weight);
}
```

lambda和方法引用实战

用不同的排序策略给一个Apple列表排序，并需要展示如何把一个原始粗暴的解决方法一步步优化。

> Java 8的API已经为你提供了一个List可用的sort方法，你不用自己去实现它。那么最困难的部分已经搞定了

但是，如何把排序策略传递给sort方法呢？你看，sort方法的签名是这样的：`void sort(Comparator<? super E> c)`。而`Comparator`是函数式接口，可以传递方法。因此我们可以认为sort的行为被参数化了。传递给它的排序策略不同，其行为也会不同。

- 首先我们的第一个解决办法可能是：

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

> 匿名内部类的方法依旧很啰嗦，因为当我们需要一种新的排序策略时，我们可能需要把上面的代码拷贝一份，但是却只需要改动`return a1.getWeight().compareTo(a2.getWeight());`这部分核心代码。策略一多便啰嗦极了。

- 用 Lambda 表达式

> 上面的例子可以看成是一个接收签名为 (T1,T2) -> int 的方法。

```
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

看起来好多了，因为lambda的类型推断，我们甚至可以1简化成下面这样：

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

> 代码还能更简洁吗？Comparator具有一个叫作comparing的静态辅助方法，
> 它可以接受一个Function来提取Comparable键值，并生成一个Comparator对象（我们会在第后面解释为什么接口可以有静态方法）。

comparing的静态辅助方法源码如下：

```
public static <T, U extends Comparable<? super U>> Comparator<T> comparing(
            Function<? super T, ? extends U> keyExtractor)
{
    Objects.requireNonNull(keyExtractor);
    return (Comparator<T> & Serializable)
        (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

它可以像下面这样用：

```
Comparator<Apple> c = Comparator.comparing((a) -> a.getWeight());
```

现在你可以把代码再改得紧凑一点了：

```
inventory.sort(Comparator.comparing((a) -> a.getWeight());
```

- 方法引用

> 前面解释过，方法引用就是替代那些转发参数的Lambda表达式的语法糖。你可以用方法引用让你的代码更简洁

```
inventory.sort(Comparator.comparing(Apple::getWeight);
```

这就是你的最终解决方案！这比Java 8之前的代码好在哪儿呢？它比较短；它的意思也很明显，并且代码读起来和问题描述差不多：“对库存进行排序，比较苹果的重量。”

> **注意：**
>
> - 这个例子中lambda表达式`Apple::getWeight`的返回值是int 因此可以采用 comparingInt方法提高内存利用率。
> - lambda表达式的返回值必须实现了Comparable接口，为可比较的元素，才能进行集合排序操作。

复合Lambda表达式

Java 8的好几个函数式接口都有为方便而设计的方法。具体而言，许多函数式接口，比如用于传递Lambda表达式的Comparator、Function和Predicate都提供了允许你进行复合的方法。

这意味着你可以把多个简单的Lambda复合成复杂的表达式。比如，你可以让两个谓词之间做一个or操作，组合成一个更大的谓词。而且，你还可以让一个函数的结果成为另一个函数的输入。

你可能会想，函数式接口中怎么可能有更多的方法呢？（毕竟，这可是违背了函数式接口的定义啊！）窍门在于，提供的允许进行复合操作的方法都是默认方法，也就是说它们不是抽象方法。

比较器复合

我们前面看到，你可以使用静态方法Comparator.comparing，如下所示：`Comparator<Apple> c = Comparator.comparing(Apple::getWeight);`

- 逆序

> 如果我们需要对之前的排序策略进行逆序怎么办？用不着去建立另一个Comparator的实例。接口有一个默认方法reversed可以使给定的比较器逆序

```
inventory.sort(comparing(Apple::getWeight).reversed());
```

- 比较器链

> 前面都很好，但如果发现有两个苹果一样重怎么办？哪个苹果应该排在前面呢？你可能需要再提供一个Comparator来进一步定义这个比较。比如，在按重量比较两个苹果之后，你可能想要按原产国排序。thenComparing方法就是做这个用的。

```
inventory.sort(comparing(Apple::getWeight)
         .reversed()
         .thenComparing(Apple::getCountry));
```

谓词复合

> 谓词接口包括三个方法：negate、and和or，你可以重用已有的Predicate来创建更复杂的谓词。比如，你可以使用negate方法来返回一个Predicate的非，比如苹果不是红的：

```
Predicate<Apple> redApple = (a) -> a.getColor().equals("red");
Predicate<Apple> notRedApple = redApple.negate();
```

> 你可能想要把两个Lambda用and方法组合起来，比如一个苹果既是红色的又比较重的：

```
Predicate<Apple> redAndHeavy = redApple.and((a) -> a.getWeight() > 150);
```

> 你可以进一步组合谓词，表达要么是重（150克以上）的红苹果，要么是绿苹果：

```
Predicate<Apple> redAndHeavyOrGreen = redApple.and((a) -> a.getWeight() > 150)
                                              .or((a) ->
a.getColor().equals("green"));
```

请注意，and和or方法是按照在表达式链中的位置，从左向右确定优先级的。因此，a.or(b).and(c)可以看作(a || b) && c。

函数复合

最后，你还可以把Function接口所代表的Lambda表达式复合起来。Function接口为此配了andThen和compose两个默认方法

> andThen方法会返回一个函数，它先对输入应用一个给定函数，再对输出应用另一个函数。比如，假设有一个函数f给数字加1 (x -> x + 1)，另一个函数g给数字乘2，你可以将它们组合成一个函数h，先给数字加1，再给结果乘2：

```
 // g(f(x)) 即：((x+1)*2)
 Function<Integer, Integer> f = x -> x + 1;
 Function<Integer, Integer> g = x -> x * 2;
 Function<Integer, Integer> h = f.andThen(g);
 int result = h.apply(1); // 返回 4
```

> 使用compose方法，先把给定的函数用作compose的参数里面给的那个函数，然后再把函数本身用于结果。比如在上一个例子里用compose的话，它将意味着f(g(x))， 而andThen则意味着g(f(x))：

```
// 数学上会写作f(g(x)) 即 ((x*2)+1)  compose组成/构成
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1); // 返回3
```

> 那么在实际中这有什么用呢？比方说你有一系列工具方法，对用String表示的一份信做文本转换：

```
public class Letter{
    public static String addHeader(String text){
        return "From caotinging: " + text;
    }
    public static String addFooter(String text){
        return text + " Kind regards";
    }
    public static String checkSpelling(String text){
        return text.replaceAll("labda", "lambda");
    }
}
```

> 现在你可以通过复合这些工具方法来创建各种转型流水线了，比如创建一个流水线：先加上抬头，然后进行拼写检查，最后加上一个落款：

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
      = addHeader.andThen(Letter::checkSpelling)
                 .andThen(Letter::addFooter);
```

2 流

流是什么

流是允许以声明性方式处理数据集合，并可以透明的并行处理，而无需写任何多线程代码。

下面两段代码都是用来返回低热量的菜肴名称的，
并按照卡路里排序，一个是用Java 7写的，另一个是用Java 8的流写的。比较一下。

java7

```
// 迭代器筛选卡路里低于400的食物
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.getCalories() < 400){
        lowCaloricDishes.add(d);
    }
}
// 进行排序
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());
    }
});
// 获取排序后低热量菜肴的名称
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
    lowCaloricDishesName.add(d.getName());
}
```

> 在这段代码中，你用了一个“垃圾变量”lowCaloricDishes。它唯一的作用就是作为一次性的中间容器。在Java 8中，实现的细节被放在它本该归属的库里了。

java8

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;

List<String> lowCaloricDishesName =
            menu.stream()
                .filter(d -> d.getCalories() < 400)
                .sorted(comparing(Dish::getCalories))
                .map(Dish::getName)
                .collect(toList());
```

> 为了利用多核架构并行执行这段代码，你只需要把stream()换成parallelStream()：

```
List<String> lowCaloricDishesName =
                menu.parallelStream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dishes::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

你可能会想，在调用parallelStream方法的时候到底发生了什么。用了多少个线程？对性能有多大提升？后面会详细讨论这些问题。现在，你可以看出，从软件工程师的角度来看，新的方法有几个显而易见的好处。

- 代码是以声明式的方式写的：说明想要完成什么，而不是说明如何实现一个操作（利用循环和if条件等控制流语句）。
- 轻松应对变化的需求：你很容易再创建一个代码版本，利用Lambda表达式来筛选高卡路里的菜肴，而用不着去复制粘贴代码。
- 你可以把几个基础操作链接起来，来表达复杂的数据处理流水线（在filter后面接上sorted、map和collect操作），同时保持代码清晰可读。filter的结果被传给了sorted方法，再传给map方法，最后传给collect方法。

别浪费太多时间了。一起来拥抱接下来介绍的强大的流吧！

总结一下，Java 8中的Stream API可以让你写出这样的代码：

- 声明式——更简洁，更易读
- 可复合——更灵活
- 可并行——性能更好

我们会使用这样一个例子：一个menu，它只是一张菜单：

```
List<Dish> menu = Arrays.asList(
 new Dish("pork", false, 800, Dish.Type.MEAT),
 new Dish("beef", false, 700, Dish.Type.MEAT),
 new Dish("chicken", false, 400, Dish.Type.MEAT),
 new Dish("french fries", true, 530, Dish.Type.OTHER),
 new Dish("rice", true, 350, Dish.Type.OTHER),
 new Dish("season fruit", true, 120, Dish.Type.OTHER),
 new Dish("pizza", true, 550, Dish.Type.OTHER),
 new Dish("prawns", false, 300, Dish.Type.FISH),
 new Dish("salmon", false, 450, Dish.Type.FISH) );
```

Dish类的定义是：

```
public class Dish {
     private final String name;
     private final boolean vegetarian; // 素
     private final int calories;
     private final Type type;

     public Dish(String name, boolean vegetarian, int calories, Type type) {
         this.name = name;
         this.vegetarian = vegetarian;
         this.calories = calories;
         this.type = type;
     }
```

```
    public String getName() {
        return name;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
    public int getCalories() {
        return calories;
    }
    public Type getType() {
        return type;
    }
    @Override
    public String toString() {
        return name;
    }
    public enum Type { MEAT, FISH, OTHER }
}
```

流简介

> 简短的定义就是“从支持数据处理操作的源生成的元素序列”。让我们一步步分析这个定义。

- **元素序列**——就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的时间/空间复杂度存储和访问元素（如ArrayList 与 LinkedList）。但流的目的在于表达计算，比如你前面见到的filter、sorted和map。集合讲的是数据，流讲的是计算。我们会在后面详细解释这个思想。
- **源**——流会使用一个提供数据的源，如集合、数组或输入/输出资源。 请注意，从有序集合生成流时会保留原有的顺序。由列表生成的流，其元素顺序与列表一致。
- **数据处理操作**——流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、sort等。流操作可以顺序执行，也可并行执行。

此外，流操作有两个重要的特点。

- **流水线**——很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的流水线。这让一些优化成为可能，如延迟和短路。流水线的操作可以看作对数据源进行数据库式查询。
- **内部迭代**——与使用迭代器显式迭代的集合不同，流的迭代操作是在背后进行的。

让我们来看一段能够体现所有这些概念的代码：

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
                    menu.stream()  // 从菜单集合中获取流-建立流水线
                        .filter(d -> d.getCalories() > 300) // 首先筛选卡路里高于300
的食物
                        .map(Dish::getName) // 获取菜名
                        .limit(3) // 截取前三个
                        .collect(toList()); // 组合成新的列表
System.out.println(threeHighCaloricDishNames);
```

> 在本例中，我们先是对menu调用stream方法，由菜单得到一个流。数据源是菜单列表，它给流提供一个元素序列。接下来，对流应用一系列数据处理操作：filter、map、limit和collect。除了collect之外，所有这些操作都会返回另一个流，这样它们就可以接成一条流水线，于是就可以看作对源的一个查询。最后，collect操作开始处理流水线，并返回结果（它和别的操作不一样，因为它返回的不是流，在这里是一个List）。在调用collect之前，没有任何结果产生，实际上根本就没有从menu里选择元素。你可以这么理解：链中的方法调用都在排队等待，直到调用collect。

过程如下所示：

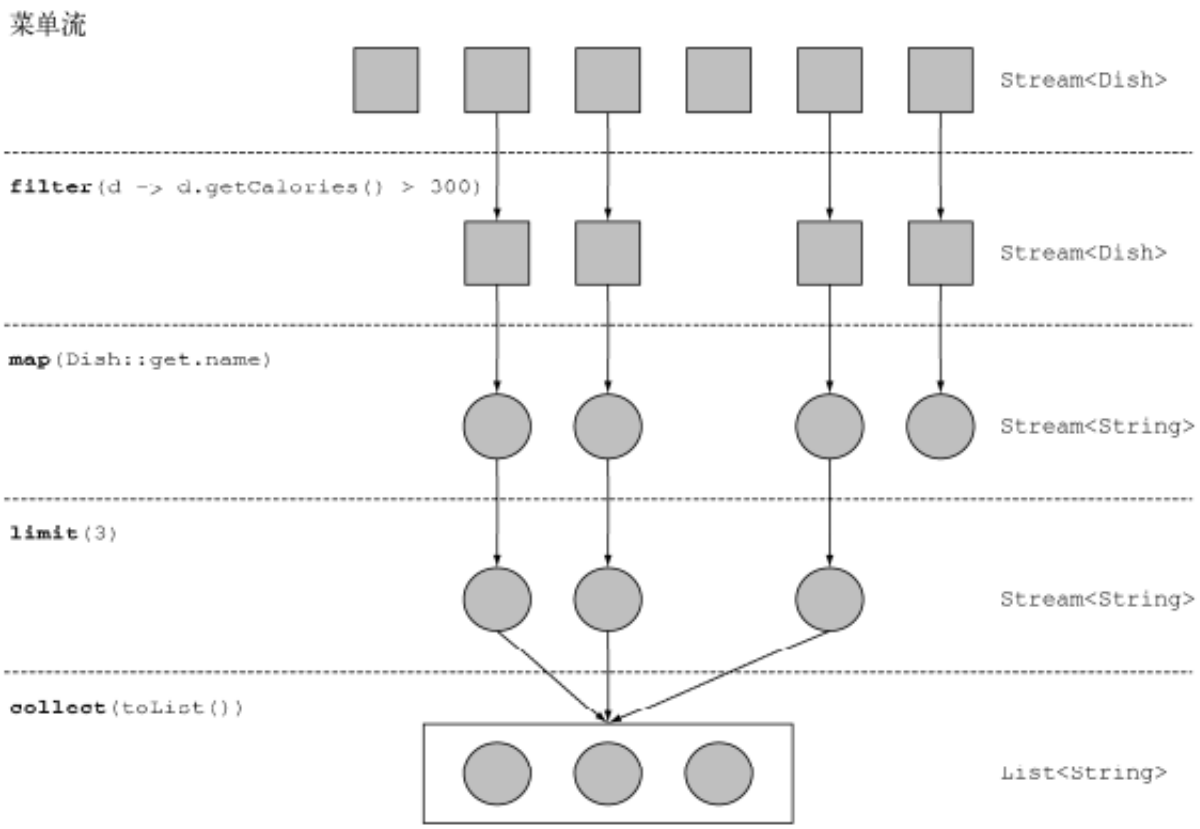


图4-2　使用流来筛选菜单，找出三个高热量菜肴的名字

流与集合

> 我们先来打个直观的比方吧。比如说存在DVD里的电影，这就是一个集合（也许是字节，也许是帧，这个无所谓），因为它包含了整个数据结构。
>
> 现在再来想想在互联网上通过视频流看同样的电影。现在这是一个流（字节流或帧流）。流媒体播放器只要提前下载用户观看位置的那几帧就可以了，这样不用等到流中大部分值计算出来，你就可以显示流的开始部分了（想想观看直播足球赛）。
>
> 特别要注意，视频播放器可能没有将整个流作为集合，保存所需要的内存缓冲区——而且要是非得等到最后一帧出现才能开始看，那等待的时间就太长了。
>
> 简单地说，集合与流之间的差异就在于什么时候进行计算。集合是一个内存中的数据结构，它包含数据结构中目前所有的值——集合中的每个元素都得先算出来才能添加到集合中。（你可以往集合里加东西或者删减东西，但是不管什么时候，集合中的每个元素都是放在内存里的，元素都得先算出来才能成为集合的一部分。）

相比之下，流则是在概念上固定的数据结构（你不能添加或删除元素），其元素则是按需计算的。 这对编程有很大的好处。在后面，我们将展示利用流构建一个质数流（2, 3, 5, 7, 11, ...）有多简单，尽管质数有无穷多个。这个思想就是用户仅仅从流中提取需要的值，而这些值——在用户看不见的地方——只会按需生成。从另一个角度来说，流就像是一个延迟创建的集合：只有在消费者要求的时候才会计算值（用管理学的话说这就是需求驱动，甚至是实时制造)。

与此相反，集合则是急切创建的（供应商驱动：先把库装满，再开始卖，就像那些昙花一现的圣诞新玩意儿一样）。以质数为例，要是想创建一个包含所有质数的集合，那这个程序算起来就没完没了了，因为总有新的质数要算，然后把它加到集合里面。当然这个集合是永远也创建不完的，消费者这辈子都见不着了。

只能遍历一次

> 请注意，和迭代器类似，流只能遍历一次。遍历完之后，我们就说这个流已经被消费掉了。你可以从原始数据源那里再获得一个新的流来重新遍历一遍，就像迭代器一样（这里假设它是集合之类的可重复的源，如果是I/O通道就没戏了）。

```
List<String> title = Arrays.asList("Java8", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
// 下面这句代码会抛出异常 java.lang.IllegalStateException：提示流已被消费
s.forEach(System.out::println);
```

所以要记得，流只能被消费一次!

内部迭代和外部迭代

> 使用Collection接口需要用户去做迭代（比如用for-each），这称为外部迭代。 相反，Streams库使用内部迭代——它帮你把迭代做了，还把得到的流值存在了某个地方，你只要给出一个函数说要干什么就可以了

```
// 集合：使用for-each循环外部迭代
List<String> names = new ArrayList<>();
for(Dish d: menu){
    names.add(d.getName());
}
```

请注意，for-each还隐藏了迭代中的一些复杂性。for-each结构是一个语法糖，它背后的东西用Iterator对象表达出来更要丑陋得多。

```
// 集合：用背后的迭代器做外部迭代
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish d = iterator.next();
    names.add(d.getName());
}
```

```
// 流：内部迭代，将得到的操作流根据提供的函数进行操作
List<String> names = menu.stream()
             .map(Dish::getName)
             .collect(toList());
```

举个例子说明：比如你希望你两岁的女儿把地上的玩具收起来

外部迭代：

```
你："我们把玩具收进盒子里，地上还有玩具吗？"
小孩："有，球。"
你："好，把球放进盒子里，还有吗？"
小孩："有，娃娃。"
你："好，把娃娃放进盒子里，还有吗？"
小孩："有，水枪。"
你："好，把水枪放进盒子里，还有吗？"
小孩："没有了"
你："好。我们收好了"
```

内部迭代：

```
// 你只需要告诉小孩，把地上的玩具放进盒子里
你："我们把地上的玩具都收进盒子里"
小孩："好"

// 小孩可以选择一只手拿球，一只手拿娃娃，一起放进盒子里，也可以先把近一点的水枪放进盒子里，效率更高
```

> 总结就是：内部迭代时，项目可以透明地并行处理，或者用更优化的顺
> 序进行处理。要是用Java过去的那种外部迭代方法，这些优化都是很困难的。
> 并且一旦通过写for-each而选择了外部迭代，那你基
> 本上就要自己管理所有的并行问题了（自己管理实际上意味着"某个良辰吉日我们会把它并行化"
> 或"开始了关于任务和synchronized的漫长而艰苦的斗争"）

流操作

stream定义了很多操作，分为两类：

```
List<String> names = menu.stream()
                .filter(d -> d.getCalories() > 300)
                .map(Dish::getName)
                .limit(3)
                .collect(toList());
```

你可以看到两类操作：
1.filter、map和limit可以连成一条流水线；
2.collect触发流水线执行并关闭它；

可以连接起来的流操作称为中间操作，关闭流的操作称为终端操作。

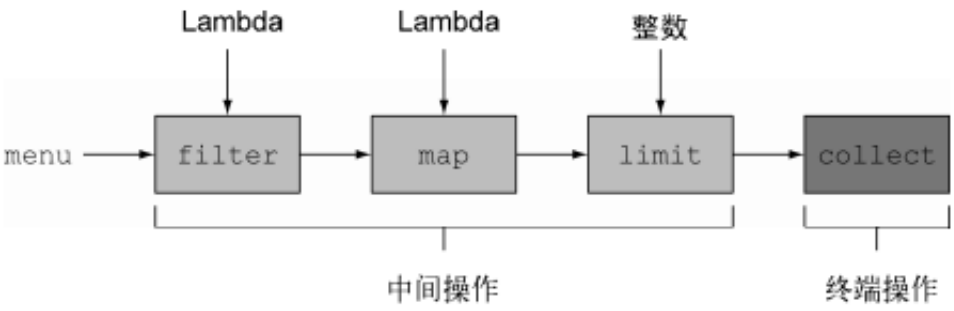


图4-5　中间操作与终端操作

中间操作

> 诸如filter或sorted等中间操作会返回另一个流。这让多个操作可以连接起来形成一个查询。重要的是，除非流水线上触发一个终端操作，否则中间操作不会执行任何处理——它们很懒。这是因为中间操作一般都可以合并起来，在终端操作时一次性全部处理。

为了搞清楚流水线中到底发生了什么，我们把代码改一改，让每个Lambda都打印出当前处理的菜肴（就像很多演示和调试技巧一样，这种编程风格要是放在生产代码里那就吓死人了，但是学习的时候却可以直接看清楚求值的顺序）：

```
List<String> names =
    menu.stream()
        .filter(d -> {
            System.out.println("filtering" + d.getName());
            return d.getCalories() > 300;
        })
        .map(d -> {
            System.out.println("mapping" + d.getName());
            return d.getName();
        })
        .limit(3)
        .collect(toList());

System.out.println(names);
```

此时打印的结果如下：

```
filtering pork
mapping pork
filtering beef
mapping beef
filtering chicken
mapping chicken
[pork, beef, chicken]
```

可以很清楚的看出来，程序并不是顺序执行的，filter、map是并行处理的。这种技术称之为循环合并。
有好几种优化利用了流的延迟性质。第一，尽管很多菜肴热量都高于300卡路里，
但只选出了前三个！这是因为limit操作以及一种称为短路的技巧。（后面会介绍）

终端操作

> 终端操作会从流的流水线生成结果。其结果是任何不是流的值，比如List、Integer，甚至void

例如，在下面的流水线中，forEach是一个返回void的终端操作，它会对源中的每道菜应用一个Lambda。把System.out.println传递给forEach，并要求它打印出由menu生成的流中的每一个Dish：

```
menu.stream().forEach(System.out::println);
```

总而言之，流的使用一般包括三件事：

1. 一个数据源（如集合）来执行一个查询；
2. 一个中间操作链，形成一条流的流水线；
3. 一个中间操作，执行流水线，并能生成结果。

下面列出了已经遇到的流操作，不涵盖全部

表4-1 中间操作

操作	类型	返回类型	操作参数	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
map	中间	Stream<R>	Function<T, R>	T -> R
limit	中间	Stream<T>		
sorted	中间	Stream<T>	Comparator<T>	(T, T) -> int
distinct	中间	Stream<T>		

表4-2 终端操作

操作	类型	目的
forEach	终端	消费流中的每个元素并对其应用 Lambda。这一操作返回 void
count	终端	返回流中元素的个数。这一操作返回 long
collect	终端	把流归约成一个集合，比如 List、Map 甚至是 Integer。详见第 6 章

> 以上都是书中所述，本人抱着实践出真知的态度亲自试了一下stream和for循环的性能比较，就在源码中的chap4中的streamBasic、结果大跌眼镜。在难以置信的情况下
> 查阅了相关资料，发现了这个：Follow-up: How fast are the Java 8 Streams?
> 这个：Java 8 Stream的性能到底如何？ 这个：Java performance tutorial – How fast are the Java 8 streams?
> 但是，很多关于集合线程安全性方面的考虑，Stream已经帮我们做了，如果在多线程场景下，java7的写法再加上一堆的同步锁等操作。结果究竟如何也不得而知。

<!--

使用流

筛选和切片

用谓词筛选

Streams接口支持filter方法（你现在应该很熟悉了）。该操作会接受一个谓词（一个返回boolean的函数）作为参数，并返回一个包括所有符合谓词的元素的流。

```
// 筛选所有素菜
List<Dish> vegetarianMenu = menu.stream()
                .filter(Dish::isVegetarian)
                .collect(toList());
```

筛选各异的元素

流还支持一个叫作distinct的方法，它会返回一个元素各异（即无重复的，根据流所生成元素的hashCode和equals方法实现）的流。例如，以下代码会筛选出列表中所有的偶数，并确保没有重复。

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
       .filter(i -> i % 2 == 0)
       .distinct()
       .forEach(System.out::println);
```

截短流

流支持limit(n)方法，该方法会返回一个不超过给定长度的流。所需的长度作为参数传递给limit。如果流是有序的，则最多会返回前n个元素。比如，你可以建立一个List，选出能量超过300卡路里的头三道菜：

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .limit(3)
                        .collect(toList());
```

跳过流

流还支持skip(n)方法，返回一个扔掉了前n个元素的流。如果流中元素不足n个，则返回一个空流。请注意，limit(n)和skip(n)是互补的！例如，下面的代码将跳过超过300卡路里的头两道菜，并返回剩下的。

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .skip(2)
                        .collect(toList());
```

映射

> 一个非常常见的数据处理套路就是从某些对象中选择信息。比如在SQL里，你可以从表中选择一列。Stream API也通过map和flatMap方法提供了类似的工具

对流中每一个元素应用函数

流支持map方法，这个方法接收一个函数作为参数，这个函数会被应用到流中的每个元素，并将其映成一个新的元素，（创建新版本而不是修改原始流）例如，下面的代码把方法引用Dish::getName传给了map方法，
来提取流中菜肴的名称：

```
List<String> dishNames = menu.stream()
                             .map(Dish::getName)
                             .collect(toList());
```

> 因为getName方法返回一个String，所以map方法输出的流的类型就是Stream。

让我们看一个稍微不同的例子来加深一下对map的理解。给定一个单词列表，你想要返回另一个列表，显示每个单词中有几个字母。怎么做呢?

你需要对列表中的每个元素应用一个函数。应用的函数应该接受一个单词，并返回其长度。你可以像下面这样，给map传递一个方法引用String::length来解决这个问题：

```
List<String> words = Arrays.asList("Java 8", "Lambdas", "In", "Action");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(toList());
```

现在让我们回到提取菜名的例子。如果你要找出每道菜的名称有多长，怎么做？你可以像下面这样，再链接上一个map：

```
List<Integer> dishNameLengths = menu.stream()
                                    .map(Dish::getName)
                                    .map(String::length)
                                    .collect(toList());
```

流的扁平化

你已经看到如何使用map方法返回列表中每个单词的长度了。让我们扩展一下：对于一张单词表，如何返回一张列表，列出里面各不相同的字符呢？例如，给定单词列表["Hello","World"]，你想要返回列表["H","e","l", "o","W","r","d"]

你可能会认为这很容易，你可以把每个单词映射成一张字符表，然后调用distinct来过滤重复的字符。第一个版本可能是这样的：

```
words.stream()
 .map(word -> word.split(""))
 .distinct()
 .collect(toList());
```

但是你会发现这根本不起作用，这个方法的问题在于，传递给map方法的Lambda（word -> word.split(""))为每个单词返回了一个String[]（String列表）。因此，map返回的流实际上是Stream<String[]>类型的。你真正想要的是用Stream来表示一个字符流。

- 尝试使用map和Arrays.stream()

首先，你需要一个字符流，而不是数组流。有一个叫作Arrays.stream()的方法可以接受一个数组并产生一个流，例如:

```
String[] arrayOfWords = {"Goodbye", "World"};
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
```

把它用在前面的那个流水线里，看看会发生什么：

```
words.stream()
 .map(word -> word.split(""))
 .map(Arrays::stream)
 .distinct()
 .collect(toList());
```

当前的解决方案仍然搞不定！这是因为，你现在得到的是一个流的列表（更准确地说是Stream类型的List）！的确，你先是把每个单词转换成一个字母数组，然后把每个数组变成了一个独立的流。

- 用flatMap

使用flatMap方法的效果是，各个数组并不是分别映射成一个流，而映射成流的内容。所有使用map(Arrays::stream)时生成的单个流都被合并起来，即扁平化为一个流。

```
List<String> uniqueCharacters = words.stream()
                                     .map(w -> w.split(""))
                                     .flatMap(Arrays::stream)
                                     .distinct()
                                     .collect(Collectors.toList());
```

> flatmap方法让你把一个流中的每个值都换成另一个流，然后把所有的流连接起来成为一个流。

查找和匹配

> 另一个常见的数据处理操作是看看数据集中的某些元素是否匹配一个给定的属性，Stream API通过allMatch、anyMatch、noneMatch、findFirst和findAny方法提供了这样的工具。

检查谓词是否至少匹配一个元素

> 标题中的谓词指的是函数式接口：Predicate T boolean

anyMatch方法可以解决 “流中是否有一个元素能匹配给定的谓词”。比如，你可以用它来看看菜单里面是否有素食可选择：

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
    System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

anyMatch方法返回一个boolean，因此是一个终端操作

检查谓词是否匹配所有元素

allMatch

allMatch方法会检查流中的元素是否都能匹配给定的谓词。比如，你可以用它来看看菜单是否有利健康（即所有菜品的热量都低于1000卡路里）：

```
boolean isHealthy = menu.stream()
    .allMatch(d -> d.getCalories() < 1000);
```

noneMatch

noneMatch它可以确保流中没有任何元素与给定的谓词匹配。比如，你可以用noneMatch重写前面的例子：

```
boolean isHealthy = menu.stream()
        .noneMatch(d -> d.getCalories() >= 1000);
```

> anyMatch、allMatch和noneMatch这三个操作都用到了我们所谓的短路，这就是大家熟悉的Java中&&和||运算符短路在流中的版本

查找元素

findAny方法将返回当前流中的任意元素。它可以与其他流操作结合使用。比如，你可能想找到一道素食菜肴。你可以结合使用filter和findAny方法来实现这个查询：

```
Optional<Dish> dish = menu.stream()
            .filter(Dish::isVegetarian)
            .findAny();
```

慢着，代码里面的Optional是个什么玩意儿?

Optional简介

Optional类（java.util.Optional）是一个容器类，代表一个值存在或不存在。在上面的代码中，findAny可能什么元素都没找到。Java 8的库设计人员引入了Optional，这样就不用返回众所周知容易出问题的null了。

- isPresent()将在Optional包含值的时候返回true, 否则返回false。
- ifPresent(Consumer block)会在值存在的时候执行给定的代码块。我们在前面介绍了Consumer函数式接口；它让你传递一个接收T类型参数，并返回void的Lambda表达式。
- T get()会在值存在时返回值，否则会抛出一个NoSuchElement异常。
- T orElse(T other)会在值存在时返回值，否则返回一个默认值。

例如，在前面的代码中你需要显式地检查Optional对象中是否存在一道素菜可以访问其名称：

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()
    .ifPresent(d -> System.out.println(d.getName()); // 如果包含值就返回菜名 否则什么都不做
```

查找第一个元素

有些流有一个出现顺序（encounter order）来指定流中项目出现的逻辑顺序（比如由List或排序好的数据列生成的流）。对于这种流，你可能想要找到第一个元素。为此有一个findFirst方法，它的工作方式类似于findany。

例如，给定一个数字列表，下面的代码能找出第一个平方能被3整除的数：

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree = someNumbers.stream()
                            .map(x -> x * x)
                            .filter(x -> x % 3 == 0)
                            .findFirst(); // 9
```

何时用findFirst和findAny

你可能会想，为什么会有findFirst和findAny呢？答案是并行。找到第一个元素在并行上限制更多。如果你不关心返回的元素是哪个，请使用findAny，因为它在使用并行流时限制更少。

归约

把一个流中的元素组合起来，使用reduce操作来表达更复杂的查询，比如“计算菜单中的总卡路里”或“菜单中卡路里最高的菜是哪一个”。

此类查询需要将流中所有元素反复结合起来，得到一个值，比如一个Integer。
用函数式编程语言的术语来说，这称为折叠（fold），因为你可以将这个操
作看成把一张长长的纸（你的流）反复折叠成一个小方块，而这就是归约操作的结果。

元素求和

在我们研究如何使用reduce方法之前，先来看看如何使用for-each循环来对数字列表中的
元素求和：

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

这段代码中有两个参数：

- 总和变量的初始值，在这里是0；
- 将列表中所有元素结合在一起的操作，在这里是+。

要是还能把所有的数字相乘，而不必去复制粘贴这段代码，不是很好？这正是reduce操
作的用武之地，它对这种重复应用的模式做了抽象。你可以像下面这样对流中所有的元素求和：

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

reduce接受两个参数：

- 一个初始值，这里是0；
- 一个BinaryOperator来将两个元素结合起来产生一个新值，这里我们用的是
  lambda (a, b) -> a + b。

你也很容易把所有的元素相乘，只需要将另一个Lambda：(a, b) -> a * b传递给reduce
操作就可以了：

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

你可以使用方法引用让这段代码更简洁。在Java 8中，Integer类现在有了一个静态的sum
方法来对两个数求和，这恰好是我们想要的，用不着反复用Lambda写同一段代码了：

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

无初始值

reduce还有一个重载的变体，它不接受初始值，但是会返回一个Optional对象：

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

> 为什么它返回一个Optional呢？考虑流中没有任何元素的情况。reduce操作无法返回其和，因为它没有初始值。这就是为什么结果被包裹在一个Optional对象里，以表明和可能不存在。

最大值和最小值

Integer类有了一个静态比较大小较大值（max）和较小值（min）的方法

- 最大值

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```

- 最小值

```
Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

> 你当然也可以写成Lambda (x, y) -> x < y ? x : y而不是Integer::min，不过后者比较易读。

> map和reduce的连接通常称为map-reduce模式，因Google用它来进行网络搜索而出名，因为它很容易并行化。在源码chap5/StreamReduce.class/ 中有实践例子

中间操作和终端操作表

表5-1　中间操作和终端操作

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

付诸实践

将迄今学到的关于流的知识付诸实践。我们来看一个不同的领域：执行交易的交易员。你的经理让你为八个查询找到答案。我在源代码给出了答案，但你应该自己先解答一下作为练习。

(1) 找出2011年发生的所有交易，并按交易额排序（从低到高）。

(2) 交易员都在哪些不同的市工作过?

(3) 查找所有来自于剑桥的交易员，并按姓名排序。

(4) 返回所有交易员的姓名字符串，按字母顺序排序。

(5) 有没有交易员是在米兰工作的?

(6) 打印生活在剑桥的交易员的所有交易额。

(7) 所有交易中，最高的交易额是多少?

(8) 找到交易额最小的交易。

以下是你要处理的领域，一个Traders和Transactions的列表：

```
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario","Milan");
Trader alan = new Trader("Alan","Cambridge");
Trader brian = new Trader("Brian","Cambridge");

List<Transaction> transactions = Arrays.asList(
     new Transaction(brian, 2011, 300),
     new Transaction(raoul, 2012, 1000),
     new Transaction(raoul, 2011, 400),
     new Transaction(mario, 2012, 710),
     new Transaction(mario, 2012, 700),
     new Transaction(alan, 2012, 950)
);
```

Trader和Transaction类的定义如下：

```
public class Trader{
    private final String name;
    private final String city;
    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }
    public String getName(){
        return this.name;
    }
    public String getCity(){
        return this.city;
    }
    public String toString(){
        return "Trader:"+this.name + " in " + this.city;
    }
}

public class Transaction{
    private final Trader trader;
    private final int year;
    private final int value;
    public Transaction(Trader trader, int year, int value){
        this.trader = trader;
        this.year = year;
        this.value = value;
    }
    public Trader getTrader(){
        return this.trader;
```

```
    }
    public int getYear(){
        return this.year;
    }
    public int getValue(){
        return this.value;
    }
    public String toString(){
        return "{" + this.trader + ", " +
        "year: "+this.year+", " +
        "value:" + this.value +"}";
    }
}
```

数值流

我们在前面看到了可以使用reduce方法计算流中元素的总和。例如，你可以像下面这样计算菜单的热量：

```
int calories = menu.stream()
                .map(Dish::getCalories)
                .reduce(0, Integer::sum);
```

这段代码的问题是，它有一个暗含的装箱成本。每个Integer都必须拆箱成一个原始类型，再进行求和。但是Stream为我们提供了更好的解决办法

原始类型流特化

Java 8引入了三个原始类型特化流接口来解决这个问题：IntStream、DoubleStream和LongStream，分别将流中的元素特化为int、long和double，从而避免了暗含的装箱成本。

映射到数值流

将流转换为特化版本的常用方法是mapToInt、mapToDouble和mapToLong。这些方法和前面说的map方法的工作方式一样，只是它们返回的是一个特化流，而不是Stream。

你可以像下面这样用mapToInt对menu中的卡路里求和：

```
int calories = menu.stream()
                .mapToInt(Dish::getCalories)
                .sum();
```

> 这里，mapToInt会从每道菜中提取热量（用一个Integer表示），并返回一个IntStream（而不是一个Stream）。然后你就可以调用IntStream接口中定义的sum方法，对卡路里求和了！请注意，如果流是空的，sum默认返回0。IntStream还支持其他的方便方法，如max、min、average等。

转换回对象流

同样，一旦有了数值流，你可能会想把它转换回非特化流。例如，IntStream上的操作只能产生原始整数： IntStream 的 map 操作接受的 Lambda 必须接受 int 并返回 int （一个 IntUnaryOperator）。但是你可能想要生成另一类值，比如Dish。为此，你需要访问Stream接口中定义的那些更广义的操作。要把原始流转换成一般流（每个int都会装箱成一个Integer），可以使用boxed方法

```
Stream<Integer> stream = intStream.boxed();
```

默认值OptionalInt

求和的那个例子很容易，因为它有一个默认值：0。但是，如果你要计算IntStream中的最大元素，就得换个法子了，因为0是错误的结果。Optional可以用Integer、String等参考类型来参数化。对于三种原始流特化，也分别有一个Optional原始类型特化版本：OptionalInt、OptionalDouble和OptionalLong。

例如，要找到IntStream中的最大元素，可以调用max方法，它会返回一个OptionalInt：

```
OptionalInt maxCalories = menu.stream()
                              .mapToInt(Dish::getCalories)
                              .max();
```

现在，如果没有最大值的话，你就可以显式处理OptionalInt去定义一个默认值了：

```
int max = maxCalories.orElse(1);
```

数值范围

假设你想要生成1和100之间的所有数值，
Java 8引入了两个可以用于IntStream和LongStream的静态方法，帮助生成这种范围：range和rangeClosed。这两个方法都是第一个参数接受起始值，第二个参数接受结束值。但range是不包含结束值的，而rangeClosed则包含结束值

```
// 这里的范围是[1,100] 结果为50个
int evenNumbers = IntStream.rangeClosed(1, 100)
                           .filter(n -> n % 2 == 0)
                           .count();

// 这里的范围是[1,100) 不包含100 结果为49
int evenNumbers2 = IntStream.range(1, 100)
                           .filter(n -> n % 2 == 0)
                           .count();
```

数值流的应用勾股数

现在我们来看一个难一点儿的例子，让你回顾一下有关数值流以及到目前为止学过的所有流操作的知识。任务就是创建一个勾股数流。

1.勾股数

那么什么是勾"数（毕达哥加斯三元数）呢？数学家毕达哥加斯发现了某些三元数(a, b, c)满足公式a * a + b * b =
c * c，其中a、b、c都是整数。例如，(3, 4, 5)就是一组有效的勾股数，因为3 * 3 + 4 * 4 = 5 * 5
或9 + 16 = 25。这样的三元数有无限组。例如，(5, 12, 13)、(6, 8, 10)和(7, 24, 25)都是有效的勾股数。勾股数很有用，因为它们描述的正好是直角三角形的三条直角边长

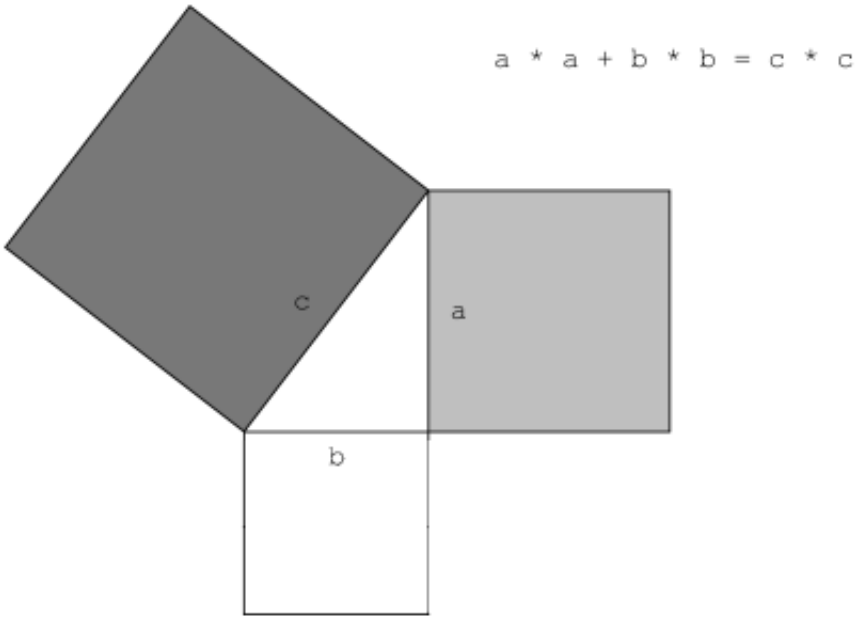


图5-9　勾股定理（毕达哥拉斯定理）

2.表示三元数

那么，怎么入手呢？第一步是定义一个三元数。虽然更恰当的做法是定义一个新的类来表示三元数，但这里你可以使用具有三个元素的int数组，比如new int[]{3, 4, 5}，来表示勾股数(3, 4, 5)。现在你就可以用数组索引访问每个元素了。

3.筛选成立的组合

假定有人为你提供了三元数中的前两个数字：a和b。怎么知道它是否能形成一组勾股数呢？你需要测试 a * a + b * b的平方根是不是整数，也就是说它没有小数部分——在Java里可以使用expr % 1表示。如果它不是整数，那就是说c不是整数。

```
filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
```

假设周围的代码给a提供了一个值，并且stream提供了b可能出现的值，filter将只选出那些可以与a组成勾股数的b。

4.生产三元组

在筛选之后，你知道a和b能够组成一个正确的组合。现在需要创建一个三元组。你可以使用map操作，像下面这样把每个元素转换成一个勾股数组：

```
stream.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
      .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

5.生产b值

前面已经看到，Stream.rangeClosed让你可以在给定区间内生成一个数值流。你可以用它来给b提供数值，这里是1到100：

```
IntStream.rangeClosed(1, 100)
    .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
    .boxed()
    .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

请注意，你在filter之后调用boxed， 从rangeClosed返回的IntStream生成一个Stream。这是因为你的map会为流中的每个元素返回一个int数组。而不是int

你也可以像下面这样使用mapToObj改写这个方法:

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .mapToObj(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

6.生成值

这里有一个关键的假设：给出了a的值。 现在，只要已知a的值，你就有了一个可以生成勾"数的流。如何解决这个问题呢？就像b一样，你需要为a生成数值

```
Stream<int[]> pythagoreanTriples =
    IntStream.rangeClosed(1, 100)
             .boxed()
             .flatMap(a -> IntStream.rangeClosed(a, 100)
                                    .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
                                    .mapToObj(b -> new int[]{a, b,
(int)Math.sqrt(a * a + b * b)})
             );
```

好的，flatMap又是怎么回事呢？首先，创建一个从1到100的数值范围来生成a的值。对每个给定的a值，创建一个三元数流。要是把a的值映射到三元数流的话，就会得到一个由流构成的流。flatMap方法在做映射的同时，还会把所有生成的三元数流扁平化成一个流。这样你就得到了一个三元数流。

还要注意，我们把b的范围改成了a到100。没有必要再从1开始了，否则就会造成重复的三元数，例如(3,4,5)和(4,3,5)。

7.运行代码

现在你可以运行解决方案，并且可以利用我们前面看到的limit命令，明确限定从生成的流中要返回多少组勾"数了：

```
pythagoreanTriples.limit(5)
                  .forEach(t -> System.out.println(t[0] + ", " + t[1] + ", " +
t[2]));
```

这会打印：

```
3, 4, 5
5, 12, 13
6, 8, 10
7, 24, 25
8, 15, 17
```

8.进一步优化

目前的解决办法并不是最优的，因为你要求两次平方根。让代码更为紧凑的一种可能的方法是，先生成所有的三元数(a*a*, *b*b, a*a*+*b*b)，然后再筛选符合条件的：

```
Stream<double[]> pythagoreanTriples2 =
            IntStream.rangeClosed(1, 100).boxed()
                     .flatMap(a -> IntStream.rangeClosed(a, 100)
                                            .mapToObj( b -> new double[]{a, b,
Math.sqrt(a*a + b*b)})
                                            .filter(t -> t[2] % 1 == 0)
                      );
```

构建流

到目前为止，你已经能够使用stream方法从集合生成流了。此外，我们还介绍了如何根据数值范围创建数值流。但创建流的方法还有许多！本节将介绍如何从值序列、数组、文件来创建流，甚至由生成函数来创建无限流！

由值创建流

你可以使用静态方法Stream.of，通过显式值创建一个流。它可以接受任意数量的参数。例如，以下代码直接使用Stream.of创建了一个字符串流。然后，你可以将字符串转换为大写，再一个个打印出来：

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

你可以使用empty得到一个空流，如下所示：

```
Stream<String> emptyStream = Stream.empty();
```

由数组创建流

你可以使用静态方法Arrays.stream从数组创建一个流。它接受一个数组作为参数。例如，你可以将一个原始类型int的数组转换成一个IntStream，如下所示：

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

由文件生成流

Java中用于处理文件等I/O操作的NIO API（非阻塞I/O）已更新，以便利用Stream API。java.nio.file.Files中的很多静态方法都会返回一个流。

> 例如，一个很有用的方法是Files.lines，它会返回一个由指定文件中的各行构成的字符串流。使用你迄今所学的内容，
> 你可以用这个方法看看一个文件中有多少各不相同的词：

```
long uniqueWords = 0;
try(Stream<String> lines = Files.lines(Paths.get("data.txt"),
Charset.defaultCharset())){
    uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
                       .distinct()
                       .count();
}
catch(IOException e){
}
```

由函数生成无限流

Stream API提供了两个静态方法来从函数生成流：Stream.iterate和Stream.generate。
这两个操作可以创建所谓的无限流：不像从固定集合创建的流那样有固定大小的流。由iterate
和generate产生的流会用**给定的函数按需创建值**，因此可以无穷无尽地计算下去！一般来说，
应该使用limit(n)来对这种流加以限制

1.迭代

我们先来看一个iterate的简单例子，然后再解释：

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

iterate方法接受一个初始值（在这里是0），还有一个依次应用在每个产生的新值上的
Lambda（UnaryOperator类型）。这里，我们使用Lambda n -> n + 2，返回的是前一个元
素加上2

> 这种iterate操作基本上是顺序的，因为结果取决于前一次。此操作将生成一个无限流——这个流没有结尾，因为值是
> 按需计算的，可以永远计算下去。这个流是无界的。

来看一个难一点儿的应用iterate的例子，如下：

> 斐波那契元组序列
>
> 斐波那契序列是经典编程练习。下面这个数列就是斐波那契序列的一部分：0, 1, 1,
> 2, 3, 5, 8, 13, 21, 34, 55...数列中开始的两个数字是0和1，后面的每个数字都是前两个数字之和。
>
> 斐波那契元组序列与此类似，是数列中数字和其后面数字组成的元组构成的序列：(0, 1),
> (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21) ...
> 你的方法是用iterate方法生成斐波那契元组序列中的前20个元素。

> 答案在这里

2.生成

与iterate方法类似，generate方法也可让你按需生成一个无限流。但generate不是依次对每个新生成的值应用函数的。它接受一个Supplier类型的Lambda提供新的值。我们先来看一个简单的用法：

```
Stream.generate(Math::random)
    .limit(5)
    .forEach(System.out::println);
```

这段代码将生成一个流，其中有五个0到1之间的随机双精度数。例如，运行一次得到了下面的结果：

```
0.9410810294106129
0.6586270755634592
0.9592859117266873
0.13743396659487006
0.3942776037651241
```

Math.Random静态方法被用作新值生成器。

我们使用的供应源（指向Math.random的方法引用）是无状态的：它不会在任何地方记录任何值，以备以后计算使用。但供应源不一定是无状态的。你可以创建存储状态的供应源，它可以修改状态，并在为流生成下一个值时使用。

> 举个例子，我们将展示如何利用generate创建斐波那契数列，这样你就可以和用iterate方法的办法比较一下。但很重要的一点是，在并行代码中使用有状态的供应源是不安全的。因此下面的代码仅仅是为了内容完整，应尽量避免使用！

> 我们在这个例子中会使用IntStream说明避免装□操作的代码。IntStream的generate方法会接受一个IntSupplier，而不是Supplier。例如，可以这样来生成一个全是1的无限流：

```
IntStream ones = IntStream.generate(() -> 1);
```

Lambda允许你创建函数式接口的实例，只要直接内联提供方法的实现就可以。你也可以像下面这样，通过实现IntSupplier接口中定义的getAsInt方法显式传递一个对象:

```
IntStream twos = IntStream.generate(new IntSupplier(){
    return 2;
    }
});
```

generate方法将使用给定的供应源，并反复调用getAsInt方法，而这个方法总是返回2。但这里使用的匿名类和Lambda的区别在于，匿名类可以通过字段定义属性（状态），而属性（状态）又可以用getAsInt方法来修改。这是一个副作用的例子。你迄今见过的所有Lambda都是没有副作用的；它们没有改

变任何状态。

回到斐波那契数列的任务上，你现在需要做的是建立一个IntSupplier，它要把前一项的值保存在属性（状态）中，以便getAsInt用它来计算下一项。此外，在下一次调用它的时候，还要更新IntSupplier的属性值。

```
IntSupplier fib = new IntSupplier(){
    private int previous = 0;
    private int current = 1;
    public int getAsInt(){
        int oldPrevious = this.previous;
        int nextValue = this.previous + this.current;

        this.previous = this.current;
        this.current = nextValue;

        return oldPrevious;
    }
};
IntStream.generate(fib).limit(10).forEach(System.out::println);
```

前面的代码创建了一个IntSupplier的实例。此对象有可变的状态：它在两个实例变量中记录了前一个斐波那契项和当前的斐波那契项。getAsInt在调用时会改变对象的状态，由此在每次调用时产生新的值。相比之下，使用iterate的方法则是纯粹不变的：它没有修改现有状态但在每次迭代时会创建新的元组。

> 你应该始终采用不改变属性（状态）的方法，以便并行处理流，并保持结果正确

用流收集数据

下面是一些查询的例子，看看你用collect和收集器能够做什么？

- 对一个交易列表按货币分组，获得该货币的所有交易额总和（返回一个Map<Currency, Integer>）
- 将交易列表分成两组：贵的和不贵的（返回一个Map<Boolean, List>）
- 创建多级分组，比如按城市对交易分组，然后进一步按照贵的或不贵的分组（返回一个Map<Boolean, List>）。

java7中你可能很快就能实现上面的需求，如下

```
Map<Currency, List<Transaction>> transactionsByCurrencies = new HashMap<>();

for (Transaction transaction : transactions) {
    Currency currency = transaction.getCurrency();
    List<Transaction> transactionsForCurrency =
transactionsByCurrencies.get(currency);

    if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();

        transactionsByCurrencies.put(currency, transactionsForCurrency);
        transactionsForCurrency.add(transaction);
    }
}
```

如果你是一位经验丰富的Java程序员，写这种东西可能挺顺手的，不过你必须承认，做这么简单的一件事就得写很多代码。更糟糕的是，读起来比写起来更费劲！代码的目的并不容易看出来，尽管换作白话的话是很直截了当的：“把列表中的交易按货币分组。”

用Stream中collect方法的一个更通用的Collector参数，你就可以用一句话实现完全相同的结果

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

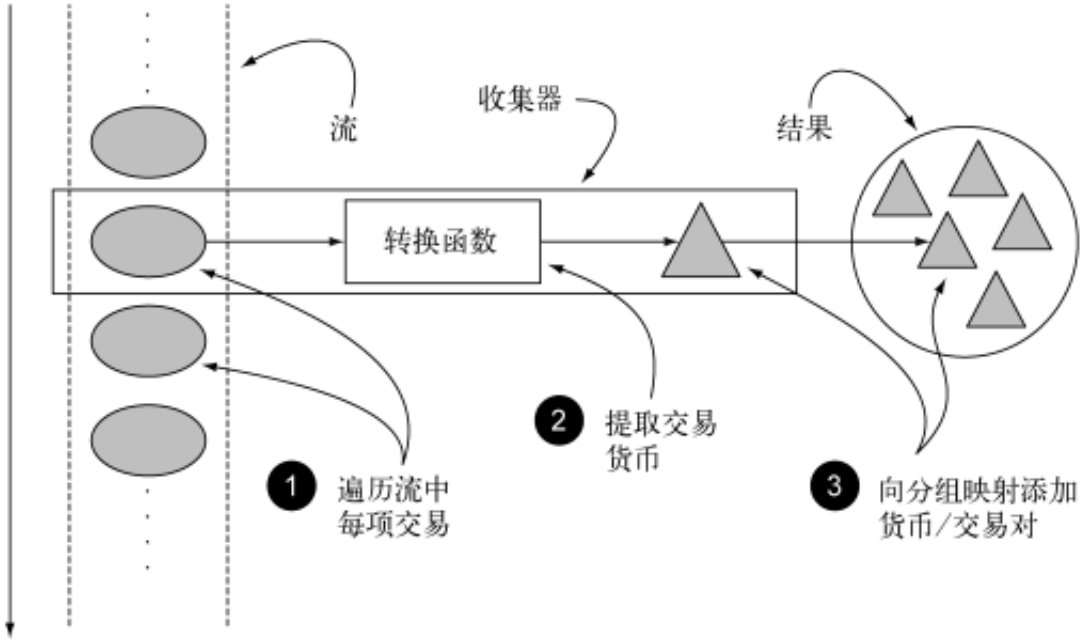
这一比差的还真多

收集器简介

指令式编程的一个主要优势：你只需指出希望的结果——“做什么”，而不用操心执行的步骤——“如何做”
传递给collect方法的参数是Collector接口的一个实现，也就是给Stream中元素做汇总的方法。
toList只是说“按顺序给每个元素生成一个列表”；在本例中，groupingBy说的是“生成一个Map，它的键(key)是（货币）桶，值(value)则是桶中那些元素的集合”

收集器用作高级归约

如图所示收集器所作所做的工作和前面java7的指令式代码一样。它遍历流中的每个元素，并让Collector进行处理



> 一般来说，Collector会对元素应用一个转换函数，并将结果存储在一个数据结构中，从而产生这一过程的最终输出。例如，在前面
> 所示的交易分组的例子中，转换函数提取了每笔交易的货币，随后使用货币作为键，将交易本身累积存储在生成的Map中。

预定义收集器

预定义收集器（即系统自带的收集器）的功能，也就是那些可以从Collectors类提供的工厂方法（例如groupingBy）创建的收集器。它们主要提供了三大功能：

- 将流元素归约和汇总为一个值
- 元素分组
- 元素分区

归约和汇总

我们先来举一个简单的例子，利用counting工厂方法返回的收集器，数一数菜单里有多少种菜：

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

这还可以写得更为直接：

```
long howManyDishes = menu.stream().count();
```

在后面的部分，我们假定你已导入了Collectors类的所有静态工厂方法：
import static java.util.stream.Collectors.*;
这样你就可以写counting()而用不着写Collectors.counting()之类的了。

查找流中的最大值和最小值

假设你想要找出菜单中热量最高的菜。你可以使用两个收集器，Collectors.maxBy和Collectors.minBy，来计算流中的最大或最小值。
这两个收集器接收一个Comparator参数来比较流中的元素。

```
Comparator<Dish> dishCaloriesComparator =
Comparator.comparingInt(Dish::getCalories);

Optional<Dish> mostCalorieDish = menu.stream()
                                     .collect(maxBy(dishCaloriesComparator));
```

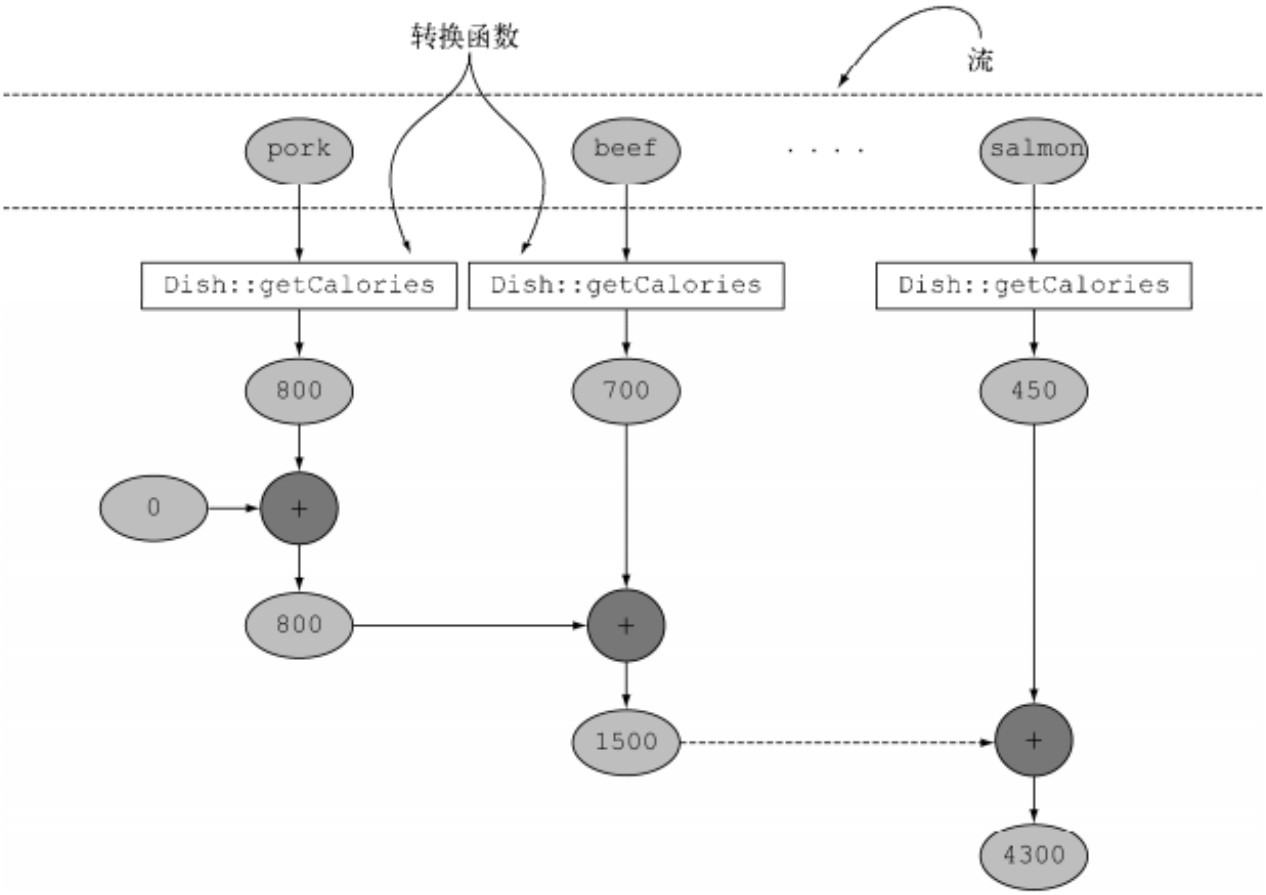
汇总

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

举个例子来说，你可以这样求出菜单列表的总热量：

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

这里的收集过程如下图所示。在遍历流时，会把每一道菜都映射为其热量，然后把这个数字累加到一个累加器里（这里的初始值0）。



> Collectors.summingLong和Collectors.summingDouble方法的作用完全一样，可以用于求和字段为long或double的情况

求平均值

汇总不仅仅是求和；还有Collectors.averagingInt，连同对应的averagingLong和averagingDouble可以计算数值的平均数：

```
// 计算菜的平均热量
double avgCalories = menu.stream().collect(averagingInt(Dish::getCalories));
```

一次性汇总操作

一次操作即可获取集合中元素的个数，总和，平均数，最大值最小值等

你可以使用summarizingInt工厂方法返回的收集器。例如，通过一次summarizing操作你可以就数出菜单中元素的个数，并得
到菜单热量总和、平均值、最大值和最小值：

```
IntSummaryStatistics menuStatistics =
menu.stream().collect(summarizingInt(Dish::getCalories));
```

这个收集器会把所有这些信息收集到一个叫作IntSummaryStatistics的类里，它提供了
方便的取值（getter）方法来访问结果。打印menuStatisticobject会得到以下输出：

```
IntSummaryStatistics{count=9, sum=4300, min=120, average=477.777778, max=800}
```

> 同样，相应的summarizingLong和summarizingDouble工厂方法有相关的LongSummaryStatistics和DoubleSummaryStatistics类型，适用于收集的属性是原始类型long或double的情况。

连接字符串

joining工厂方法返回的收集器会把对流中每一个对象应用toString方法得到的所有字符串连接成一个字符串。这意味着你把菜单中所有菜单的名称连接起来，如下所示：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

oining在内部使用了StringBuilder来把生成的字符串逐个追加起来。此外，如果Dish类的toString方法返回菜单的名称，那你无需用提取每一道菜的名称的
函数来对原流做映射就能够得到相同的结果：

```
String shortMenu = menu.stream().collect(joining());
```

两者均可产生字符串: porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon

该字符串的可读性并不好。joining工厂方法有一个重载版本可以接受元素之间的
分界符，这样你就可以得到一个带有逗号分隔的菜肴名称列表：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

它会生成：pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon

广义的归约汇总

事实上，我们已经讨论的所有收集器，都可以用reducing工厂方法Collectors.reducing工厂方法实现。可以说，前面讨论的只是提高代码可读性。

例如，可以用reducing方法创建的收集器来计算菜单的总热量，如下所示：

```
int totalCalories = menu.stream().collect(reducing( 0, Dish::getCalories, (i, j)
-> i + j));
```

它需要三个参数:

- 第一个参数是归约操作的起始值，也是流中没有元素时的返回值.
- 第二个参数就是收集操作中的转换函数。

- 第三个参数是一个BinaryOperator，将上一次操作返回的值和当前元素收集成一个同类型的值

同样，你可以使用下面这样单参数形式的reducing来找到热量最高的菜，如下所示：

```
Optional<Dish> mostCalorieDish = menu.stream()
          .collect(reducing((d1, d2) -> d1.getCalories() > d2.getCalories() ? d1
: d2));
```

reduce和collect的区别

reduce和collect方法通常可以获得同样的结果，例如，你可以用如下的方式用reduce方法来实现toListCollector所做的工作：

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
List<Integer> numbers = stream.reduce(
        new ArrayList<Integer>(),(List<Integer> l, Integer e) -> {
                                                    l.add(e);
                                                    return l;
                                },
                (List<Integer> l1, List<Integer> l2) -> {
                                                l1.addAll(l2);
                                                return l1;
        });
```

这个解决方法有两个问题：

- reduce方法旨在把两个值结合起来生成一个新值，它是一个不可变的归约。
- collect方法的设计就是要改变容器，累积出要的结果。

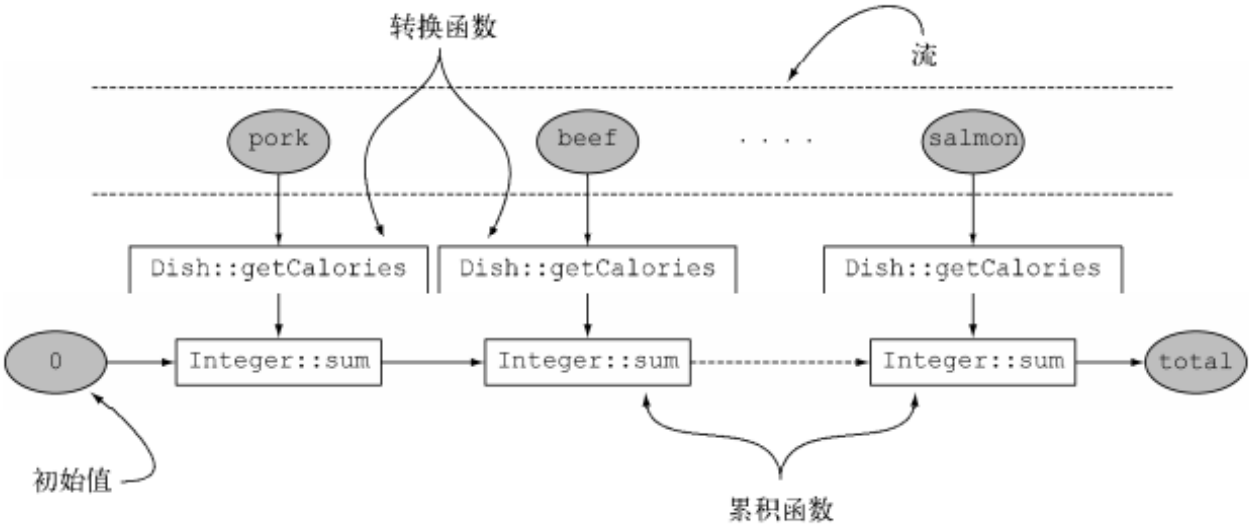
这意味着，上面的代码是在滥用reduce方法，因为它在原地改变了作为容器的List。

错误的使用reduce方法还会造成一个实际问题：这个归约过程不能并行工作，因为由多个线程并发修改同一个数据结构可能会破坏List本身。在这种情况下，想要线程安全，就必须要每次分配一个新的List，而对象分配又会影响性能。
这就是collect方法更适合表达可变容器上的归约的原因，它更适合并行操作。

1. 收集器的灵活性：以不同的方法执行同样的操作

你还可以进一步简化前面使用reducing收集器的求和例子——引用Integer类的sum方法

```
// reducing(初始值,转换函数,累积函数)
int totalCalories =
menu.stream().collect(reducing(0,Dish::getCalories,Integer::sum));
```



counting收集器也是类似地利用三参数reducing工厂方法实现的。它把流中的每个元素都转换成一个值为1的Long型对象，然后再把它们相加：

```
public static <T> Collector<T, ?, Long> counting() {
    return reducing(0L, e -> 1L, Long::sum);
}
```

使用泛型？ 通配符

在刚刚提到的代码片段中，？ 通配符它用作counting工厂方法返回的收集器签名中的第二个泛型类型。在这里，
它仅仅意味着收集器的累加器类型未知，即，累加器本身可以是任何类型。

还有另一种方法不使用收集器也能执行相同操作

```
int totalCalories =
menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

更简洁的方法是把流映射到一个IntStream，然后调用sum方法：

```
int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();
```

2. 根据情况选择最佳解决方案

函数式编程通常提供了多种方法来执行同一个操作.
收集器在某种程度上比Stream接口上直接提供的方法用起来更复杂，但好处在于它们能提供更高水平的抽象和概
括，也更容易重用和自定义。

> 尽可能为问题探索不同的解决方案，但在通用的方案里面，始终选择最专门化的一个。无论是从可读性还是性能上看，
> 这一般都是最好的决定。例如，要计算菜单的总热量，我们更倾向于最后一个解决方案（使用IntStream），因为它最简明，也很可能最易读。
> 同时，它也是性能最好的一个，因为IntStream可以让我们避免自动装箱操作。

分组

假设你要把菜单中的菜肴按照类型进行分类，有鱼的放一组，有肉的放一组，其他的都放另一组。用Collectors.groupingBy工厂方法返回
的收集器就可以轻松地完成这项任务：

```
Map<Dish.Type, List<Dish>> dishesByType =
menu.stream().collect(groupingBy(Dish::getType));
```

其结果是下面的map：

```
{
    FISH=[prawns, salmon],
    OTHER=[french fries, rice, season fruit, pizza],
    MEAT=[pork, beef, chicken]
}
```

这里，你给groupingBy方法传递了一个Function（以方法引用的形式），它提取了流中每一道Dish的Dish.Type。这个Function叫作分类函数，用来把流中的元素分成不同的组。

> 你想用以分类的条件可能比简单的属性访
> 问器要复杂。例如，你可能想把热量不到400卡路里的分为“低ㅋ量”（diet），400到700卡路里的分为“普通”（normal），高于700卡路里的为“高热量”（fat）。由于Dish类的作者
> 没有把这个操作写成一个方法，你无法使用方法引用，但你可以把这个逻辑写成Lambda表达式：

```
public enum CaloricLevel { DIET, NORMAL, FAT }

Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream()
        .collect(
            groupingBy(dish -> {
                if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            })
 );
```

你已经看到了如何对菜单按照类型和热量进行分组，但要是想同时按照这两个标准分类怎么办呢？分组的强大之处就在于它可以有效地组合

多级分组

要实现多级分组，我们可以使用一个由双参数版本的Collectors.groupingBy工厂方法创建的收集器，它除了普通的分类函数之外，还可以接受collector类型的第二个参数。那么要进行二级分组的话，我们可以把一个内层groupingBy传递给外层groupingBy，并定义一个为流中项目分类的二级标准

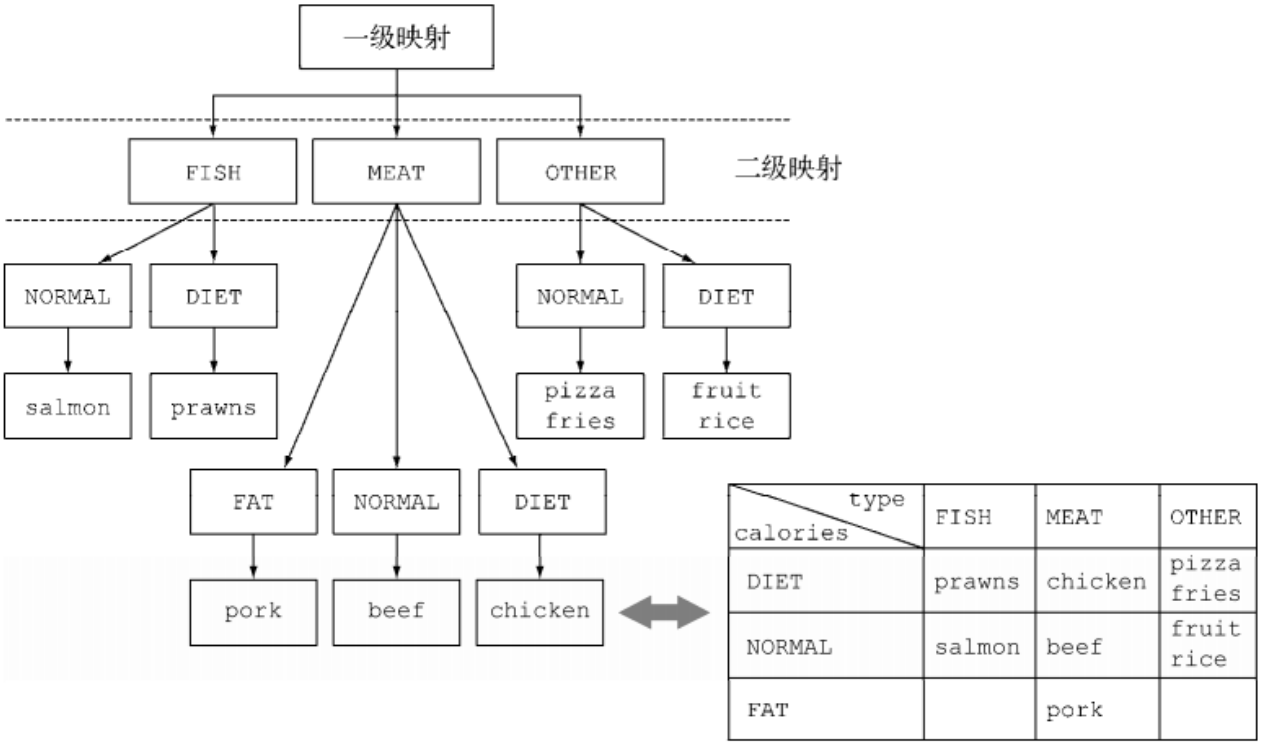
```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
    menu.stream().collect(
        groupingBy(Dish::getType,
                        groupingBy(dish -> {
                                    if (dish.getCalories() <= 400) return
CaloricLevel.DIET;
                                    else if (dish.getCalories() <= 700) return
CaloricLevel.NORMAL;
                                    else return CaloricLevel.FAT;
                                } )
                )
);
```

这个二级分组的结果就是像下面这样的两级Map：

```
{
    MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
    FISH={DIET=[prawns], NORMAL=[salmon]},
    OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}
}
```

这里的外层Map的键就是第一级分类函数生成的值：“fish, meat, other”，而这个Map的值又是一个Map，键是二级分类函数生成的值：“normal, diet, fat”。

最后，第二级map的值是流中元素构成的List，是分别应用第一级和第二级分类函数所得到的对应第一级和第二级键的值：“salmon、pizza...”这种多级分组操作可以扩展至任意层级，n级分组就会得到一个代表n级树形结构的n级Map。



calories \ type	FISH	MEAT	OTHER
DIET	prawns	chicken	pizza fries
NORMAL	salmon	beef	fruit rice
FAT		pork	

图6-5　*n*层嵌套映射和*n*维分类表之间的等价关系

按子组收集数据

传递给第一个groupingBy的第二个收集器可以是任何类型，要数一数菜单中每类菜有多少个，可以传递counting收集器作为
groupingBy收集器的第二个参数：

```
Map<Dish.Type, Long> typesCount = menu.stream().collect(groupingBy(Dish::getType,
counting()));
```

其结果是下面的Map：

```
{MEAT=3, FISH=2, OTHER=4}
```

还要注意，普通的单参数groupingBy(f)（其中f是分类函数）实际上是groupingBy(f, toList())的简便写法

例如：
按照菜的类型分类获取每个类型下最高热量的菜：

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
     menu.stream()
         .collect(groupingBy(Dish::getType,
maxBy(comparingInt(Dish::getCalories))));
```

结果如下：

```
{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}
```

> 注意：
> 这个Map中的值是Optional，因为这是maxBy工厂方法生成的收集器的类型，但实际上，如果菜单中没有某一类型的Dish，这个类型就不会对应一个Optional.empty()值，而且根本不会出现在Map的键中。groupingBy收集器只有在应用分组条件后后，第一次在流中找到某个键对应的元素时才会把该键加入分组Map中。这意味着Optional包装器在这里不是很有用。

1.把收集器的结果转换为另一种类型

因为分组操作的Map结果中的每个值上包装的Optional没什么用，所以你可能想要把它们去掉，简言之就是把收集器返回的结果转换为另一种类型，你可以使用Collectors.collectingAndThen工厂方法返回的收集器，如下所示。

```
Map<Dish.Type, Dish> mostCaloricByType =
        menu.stream()
            .collect(groupingBy(Dish::getType,
                        collectingAndThen(
                                maxBy(comparingInt(Dish::getCalories)),
                                Optional::get)));
```

这个工厂方法接受两个参数——要转换的收集器以及转换函数，并返回另一个收集器。这个收集器相当于旧收集器的一个包装，collect操作的最后一步就是将返回值用转换函数做一个映射。在这里，被包起来的收集器就是用maxBy建立的那个，而转换函数Optional::get则把返回的Optional中的值提取出来

结果如下:

```
{FISH=salmon, OTHER=pizza, MEAT=pork}
```

2.与groupingBy联合用的其他收集器的例子

例如，还可以联合使用对所有同类型的菜肴热量求和的收集器：

```
Map<Dish.Type, Integer> totalCaloriesByType =
                        menu.stream().collect(groupingBy(Dish::getType,

 summingInt(Dish::getCalories)));
```

常常和groupingBy联合使用的另一个收集器是mapping方法.

这个方法接受两个参数：一个函数对流中的元素做变换，另一个则将变换的结果对象收集起来

我们来看一个使用这个收集器的实际例子。比方说你想要知道，对于每种类型的Dish，菜单中都有哪些CaloricLevel。我们可以把groupingBy和mapping收集器结合起来，如下所示：

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
    menu.stream().collect(
            groupingBy(Dish::getType, mapping(
                                dish -> {
                                    if (dish.getCalories() <= 400) return
CaloricLevel.DIET;
                                        else if (dish.getCalories() <= 700)
return CaloricLevel.NORMAL;
                                            else return CaloricLevel.FAT; },
                                toSet())));
```

这里，就像我们前面见到过的，传递给映射方法的转换函数将Dish映射成了它的CaloricLevel：生成的CaloricLevel流传递给一个toSet收集器，以便仅保留各不相同的值。得到这样的Map结果：

```
{OTHER=[DIET, NORMAL], MEAT=[DIET, NORMAL, FAT], FISH=[DIET, NORMAL]}
```

> 请注意在上一个示例中，对于返回的Set是什么类型并没有任何保证。但通过使用toCollection，你就可以有更多的控制。例如，你可以给它传递一个构造函数引用来要求HashSet：

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
  menu.stream().collect(
    groupingBy(Dish::getType, mapping(
                dish -> { if (dish.getCalories() <= 400) return
CaloricLevel.DIET;
                          else if (dish.getCalories() <= 700) return
CaloricLevel.NORMAL;
                              else return CaloricLevel.FAT; },
                      toCollection(HashSet::new) )));
```

分区

分区是分组的特殊情况：由一个谓词（返回一个布尔值的函数）作为分类函数，它称分区函数。分区函数返回一个布尔值，这意味着得到的分组Map的键类型是Boolean，于是它最多可以分为两组——true是一组，false是一组。例如，你想将菜单中素食和非素食分开：

```
Map<Boolean, List<Dish>> partitionedMenu =
    menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

这会返回下面的Map：

```
{
    false=[pork, beef, chicken, prawns, salmon],
    true=[french fries, rice, season fruit, pizza]
}
```

那么通过Map中键为true的值，就可以找出所有的素菜了：

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

用同样的分区谓词，对菜单List创建的流作筛选，然后把结果收集到另外一个List
中也可以获得相同的结果：

```
List<Dish> vegetarianDishes =
    menu.stream().filter(Dish::isVegetarian).collect(toList());
```

分区的优势

分区的好处在于保留了分区函数返回true或false的两套流元素列表。在上一个例子中，当你想获得素食和非素食两个集合时。
分区函数的优势就表现出来了。

并且partitioningBy工厂方法有一个重载版本，可以像下面这样传递第二个收集器：

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
        menu.stream().collect(
                        partitioningBy(Dish::isVegetarian,
                                    groupingBy(Dish::getType)));
```

这将产生一个二级Map：

```
{
    false={
            FISH=[prawns, salmon],
            MEAT=[pork, beef, chicken]
            },
    true={
            OTHER=[french fries, rice, season fruit, pizza]
    }
}
```

这里，对于分区产生的素食和非素食子流，分别按类型对类型分组，得到了一个二级Map。
再举一个例子，你可以重用前面的代码来找到素食和非素食中热量最高的菜：

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
    menu.stream().collect(
            partitioningBy(Dish::isVegetarian,
                collectingAndThen(
                    maxBy(comparingInt(Dish::getCalories)),
                        Optional::get)));
```

这将产生以下结果：

```
{false=pork, true=pizza}
```

将数字按质数和非质数区分

假设你要写一个方法，它接受参数int n，并将前n个自然数分为质数和非质数。
首先，编写能测试一个int是否为质数的谓词函数

```
public boolean isPrime(int candidate) {
    //判断 从2开始到参数为止（不包括本身）的所有数是否都不能被该参数整除，如果都不能返回true--为质数
    return IntStream.range(2, candidate).noneMatch(i -> candidate % i == 0);
}
```

简单的优化一下上述函数：

```
public boolean isPrime(int candidate) {
    // 仅测试小于等于待测数平方根的因子：
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot).noneMatch(i -> candidate % i == 0);
}
```

现在最主要的一部分工作已经做好了。为了把前n个数字分为质数和非质数，只要创建一个包含这n个数的流，用已经写好的isPrime方法作为谓词，再给partitioningBy收集器收集归约就好了：

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
                                    .collect(partitioningBy(candidate -> isPrime(candidate)));
}
```

现在我们已经讨论过了Collectors类的静态工厂方法能够创建的所有收集器，并介绍了使用它们的实际例子。下表将它们汇总到一起，给出了它们应用到Stream上返回的类型，以及它们用于一个叫作menuStream的Stream上的实际例子。

Collectors收集器汇总表

表6-1 Collectors类的静态工厂方法

工厂方法	返回类型	用 于
toList	List<T>	把流中所有项目收集到一个 List
使用示例：List<Dish> dishes = menuStream.collect(toList());		
toSet	Set<T>	把流中所有项目收集到一个 Set，删除重复项
使用示例：Set<Dish> dishes = menuStream.collect(toSet());		
toCollection	Collection<T>	把流中所有项目收集到给定的供应源创建的集合
使用示例：Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);		
counting	Long	计算流中元素的个数
使用示例：long howManyDishes = menuStream.collect(counting());		
summingInt	Integer	对流中项目的一个整数属性求和

工厂方法	返回类型	用 于
使用示例：int totalCalories = menuStream.collect(summingInt(Dish::getCalories));		
averagingInt	Double	计算流中项目 Integer 属性的平均值
使用示例：double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));		
summarizingInt	IntSummaryStatistics	收集关于流中项目 Integer 属性的统计值，例如最大、最小、总和与平均值
使用示例：IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories));		
joining`	String	连接对流中每个项目调用 toString 方法所生成的字符串
使用示例：String shortMenu = menuStream.map(Dish::getName).collect(joining(", "));		
maxBy	Optional<T>	一个包裹了流中按照给定比较器选出的最大元素的 Optional，或如果流为空则为 Optional.empty()
使用示例：Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));		
minBy	Optional<T>	一个包裹了流中按照给定比较器选出的最小元素的 Optional，或如果流为空则为 Optional.empty()
使用示例：Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories)));		
reducing	归约操作产生的类型	从一个作为累加器的初始值开始，利用 BinaryOperator 与流中的元素逐个结合，从而将流归约为单个值
使用示例：int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));		
collectingAndThen	转换函数返回的类型	包裹另一个收集器，对其结果应用转换函数
使用示例：int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));		
groupingBy	Map<K, List<T>>	根据项目的一个属性的值对流中的项目作问组，并将属性值作为结果 Map 的键
使用示例：Map<Dish.Type,List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));		
partitioningBy	Map<Boolean,List<T>>	根据对流中每个项目应用谓词的结果来对项目进行分区
使用示例：Map<Boolean,List<Dish>> vegetarianDishes =		

```
menuStream.collect(partitioningBy(Dish::isVegetarian));
```

收集器接口

自定义Collector接口，首先需要了解源码结构：

```
//Collector接口
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

本列表适用以下定义。

- T是流中要收集的项目的泛型。
- A是累加器的类型，累加器是在收集过程中用于收集部分结果的对象。
- R是收集操作得到的对象（通常但并不一定是集合）的类型。

例如，你可以实现一个ToListCollector类，将Stream中的所有元素收集到一个List里，它的签名如下：
`public class ToListCollector<T> implements Collector<T, List<T>, List<T>>`

这里用于累积的对象也将是收集过程的最终结果。

理解收集器接口的方法

1、建立新的结果容器：supplier方法

supplier方法必须返回一个结果为空的Supplier，也就是一个无参数函数，在调用时它会
创建一个空的累加器实例，供数据收集过程使用：

```
public Supplier<List<T>> supplier() {
    return () -> new ArrayList<T>();
}

//也可以只传递一个构造函数引用：
public Supplier<List<T>> supplier() {
    return ArrayList::new;
}
```

2、将元素添加到结果容器：accumulator方法

accumulator方法会返回执行累加操作的函数。当遍历到流中第n个元素时，这个函数执行时会有两个参数：保存累加结果的累加器（已收集了流中的前 n-1 个项目），还有第n个元素本身。

```
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```

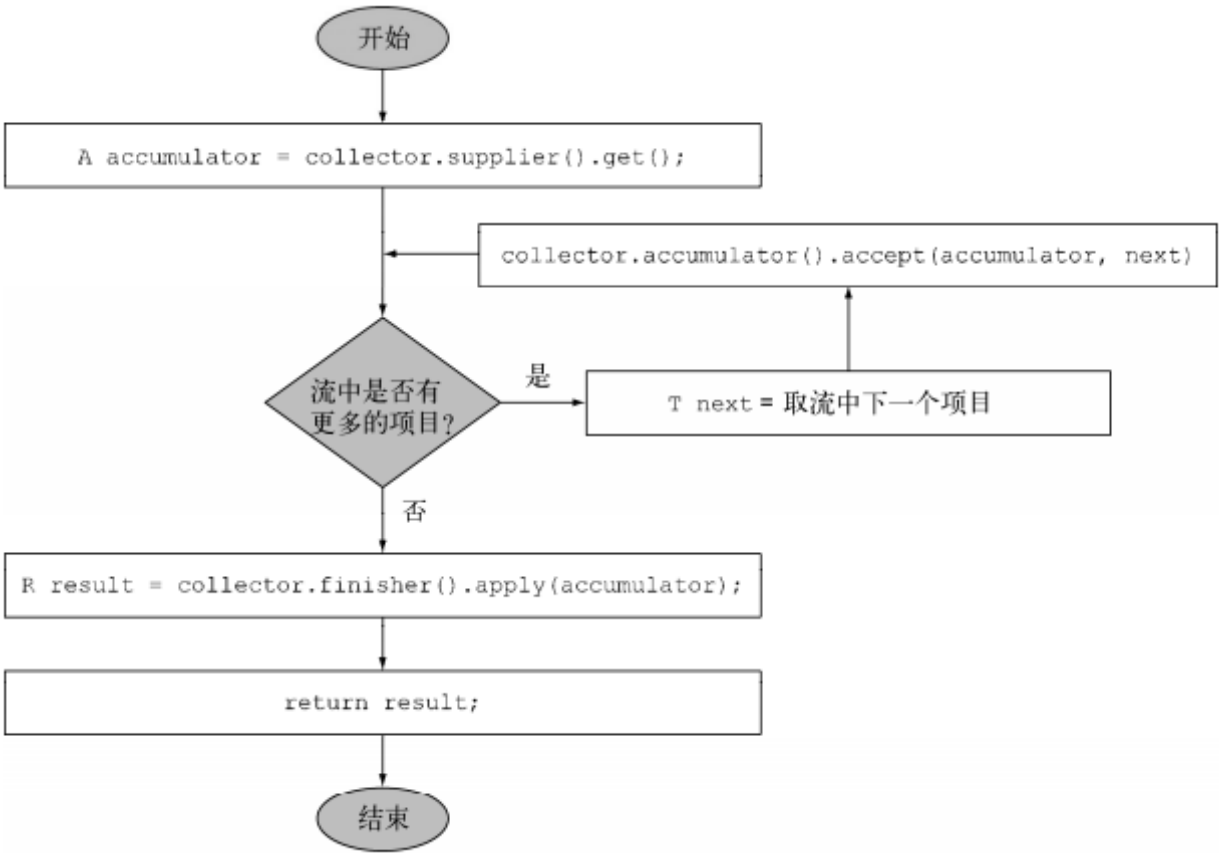
3、对结果容器用最终转换：finisher方法

在遍历完流后，finisher方法必须返回在ℨ㐅过程的最后要调用的一个函数，以便将ℨ加器对象转换为整个集合操作的最终结果：

像ToListCollector的情况一样，累加器对象刚好符合预期的最终结果，因此无需进行转换。所以finisher方法只需返回identity函数：

```
public Function<List<T>, List<T>> finisher() {
 return Function.identity();
}
```

> 通常这三个方法就足以对流进行顺序归约，但是往往需要对流进行并行操作。这三个方法的归约流程大概如下：



4、合并多个结果容器：combiner方法

四个方法中的最后一个——combiner方法会返回一个供归约操作使用的函数，它定义了对流的各个子部分进行并行处理时，各个子部分归约所得的累加器要如何合并。对于toList而言，这个方法的实现非常简单，只要把从流的第二个部分收集到的项目列表加到遍历第一部分时得到的列表后面就行了：

```
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1;
    }
}
```

- 原始流会以递归方式划分为子流，直到定义流是否需要进一步划分的一个条件为非（如果分布式工作单位太小，并行计算往往比顺序计算要慢，而且要是生成的并行任务比处理器内核数多很多的话就毫无意义了）。
- 现在，所有的子流都可以并行处理，即对每个子流应用顺序归约算法。
- 最后，使用收集器combiner方法返回的函数，将所有的部分结果两两合并。这时会把原始流每次划分时得到的子流对应的归约结果合并起来。

5、characteristics方法

最后一个方法——characteristics会返回一个不可变的Characteristics集合，它定义了收集器的行为——尤其是关于流是否可以并行归约，以及可以使用哪些优化的提示。Characteristics是一个包含三个项目的枚举。

- UNORDERED——归约结果不受流中项目的遍历和累积顺序的影响。
- CONCURRENT——accumulator函数可以从多个线程同时调用，且该收集器可以并行归约流。如果收集器没有标为UNORDERED，那它仅在用于无序数据源时才可以并行归约。
- IDENTITY_FINISH——这表明完成器方法返回的函数是一个恒等函数，可以跳过。这种情况下，累加器对象将会直接用作归约过程的最终结果。这也意味着，将累加器A不加检查地转换为结果R是安全的。

> 我们迄今开发的ToListCollector是IDENTITY_FINISH的，因为用来累积流中元素的List已经是我们要的最终结果，用不着进一步转换了，但它并不是UNORDERED，因为用在有序流上的时候，我们还是希望顺序能够保留在得到的List中。最后，它是CONCURRENT的，但我们说过了，仅仅在背后的数据源无序时才会并行处理。

融合完整的ToListCollector

融合前面所介绍的方法实现，足够我们开发自己的toList收集器了，代码如下：

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import static java.util.stream.Collector.Characteristics.*;
```

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {

    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;
    }

    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;
    }

    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.indentity();
    }

    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }

    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(
        IDENTITY_FINISH, CONCURRENT));
    }
}
```

> 这个实现与Collectors.toList方法并不完全相同，但区别仅仅是一些小的优化。
> ava API所提供的收集器在需要返回空列表时使用了Collections.emptyList()这个单例（singleton）。

```
List<Dish> dishes = menuStream.collect(new ToListCollector<Dish>());
```

这个实现和标准的

```
List<Dish> dishes = menuStream.collect(toList());
```

构造之间的其他差异在于toList是一个工厂，而ToListCollector必须用new来实例化

进行自定义收集而不实现Collector接口

对于IDENTITY_FINISH的收集操作，还有一种方法可以得到同样的结果而无需从头实现新的Collectors接口.

Stream有一个重载的collect方法可以接受另外三个函数——supplier、accumulator和combiner，其语义和Collector接口的相应方法返回的函数完全相同

所以比
如说，我们可以像下面这样把菜肴流中的项目收集到一个List中：

```
List<Dish> dishes = menuStream.collect(
    ArrayList::new, // 供应源
    List::add,      // 累加器
    List::addAll    // 并行组合器
 );
```

这第二种形式虽然比前一个写法更为紧凑和简洁，却不那么易读。此外，以恰当的类来实现自己的自定义收集器有助于重用并可避免代码重复。另外这第二个collect方法不能传递任何Characteristics，所以它永远都是一个IDENTITY_FINISH和CONCURRENT但并非UNORDERED的收集器。

开发自定义质数收集器

讨论分区的时候，我们用Collectors类提供的一个方便的工厂方法创建了一个收集器，它将前n个自然数划分为质数和非质数，如下所示。

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
                .collect(partitioningBy(candidate -> isPrime(candidate));
}

public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                .noneMatch(i -> candidate % i == 0);
}
```

还有没有办法来获得更好的性能呢？当然，开发一个自定义收集器即可

自定义质数收集器

一个可能的优化是仅仅看看被测试数是不是能够被质数整除。要是除数本身都不是质数就用不着测了。所以我们可以仅仅用被测试数之前的质数来测试。完整例子如下：

```
/**
* T: Integer 收集的元素类型为Integer
* A: Map<Boolean, List<Integer>> 累加器收集结果的类型。
* R: Map<Boolean, List<Integer>> 收集操作最后的结果类型
*/
public class PrimeNumbersCollector implements Collector<Integer, Map<Boolean,
List<Integer>>, Map<Boolean, List<Integer>>> {

    /**
    * 创建用作累加器的容器Map。初始化true false两个键的空列表，分别用来收集质数和非质数
    * @return
    */
    @Override
    public Supplier<Map<Boolean, List<Integer>>> supplier() {
        return () -> new HashMap<Boolean, List<Integer>>() {{
                            put(true, new ArrayList<Integer>());
                            put(false, new ArrayList<Integer>());
        }};
    }

    /**
    * 实现累加过程：通过调用isPrime函数判断元素是否为质数，并添加到相应的键中
    * @return
    */
    @Override
    public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
        return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
                    acc.get( isPrime( acc.get(true), candidate) )
                       .add(candidate);
        };
    }

    /**
    * 让收集器并行工作：请注意这个收集器实际上是无法并行工作的，因为isPrime的原因，该算法本身就
是顺序的，意味着
    * 这个combiner方法永远都不会被调用，这里为了示例还是补全方法，
    * 实际中可以留空，也可以抛出UnsupporteOperationException异常
    * @return
    */
    @Override
    public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
        return (Map<Boolean, List<Integer>> map1, Map<Boolean, List<Integer>>
map2) -> {
                    map1.get(true).addAll(map2.get(true));
                    map1.get(false).addAll(map2.get(false));
                    return map1;
```

```
        };
    }

    /**
    * 因为累加结果就是最后的结果，因此这里也不需要做其他的转化操作。
    * @return
    */
    @Override
    public Function<Map<Boolean, List<Integer>>, Map<Boolean, List<Integer>>>
finisher() {
        return Function.identity();
    }

    /**
    * 收集器特征值
    * @return
    */
    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
    }
}
```

也可以通过collect的重载版本实现这个收集器，如下：

```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector (int n) {
    IntStream.rangeClosed(2, n).boxed().collect(
            () -> new HashMap<Boolean, List<Integer>>() {{
                            put(true, new ArrayList<Integer>());
                            put(false, new ArrayList<Integer>());
            }},
            (acc, candidate) -> { acc.get( isPrime(acc.get(true), candidate) )
                                    .add(candidate);
            },
            (map1, map2) -> {
                    map1.get(true).addAll(map2.get(true));
                    map1.get(false).addAll(map2.get(false));
            }
    );
}
```

你看，这样就可以避免为实现Collector接口创建一个全新的类；得到的代码更紧凑，虽然可能可读性会差一点，可重用性会差一点。

并行处理数据与性能

在Java 7之前，并行处理数据集合非常麻烦。

- 第一，你得明确地把包含数据的数据结构分成若干子部分。
- 第二，你要给每个子部分分配一个独立的线程。
- 第三，你需要在恰当的时候对它们进行同步来避免不希望出现的竞争条件，等待所有线程完成，最后把这些部分结果合并起
  来。

Java 7引入了一个叫作分支/合并的框架，让这些操作更稳定、更不易出错。

Stream接口可以让你不用太费力气就能对数据集执行并行操作。它允
许你声明性地将顺序流变为并行流。此外你还将看到
流是如何在幕后应用Java 7引入的分支/合并框架的。你还会发现，了解并行流内部是如何工作的
很重要，因为如果你如果误用这一方面，就可能因误用而得到意外的（很可能是错的）结果。

并行流

在前面。我们简要地提到了Stream接口可以让你非常方便地处理它的元素：可以通过对
收集源调用parallelStream方法来把集合转换为并行流。并行流就是一个把内容分成多个数据
块，并用不同的线程分别处理每个数据块的流。这样一来，你就可以自动把给定操作的工作负荷
分配给多核处理器的所有内核，让它们都动起来。让我们用一个简单的例子来体验一下这个思想。

假设你需要写一个方法，接受数字n作为参数，并返回从1到给定参数的所有数字的和。一个
直接的方法是生成一个数字流，把它限制到给定的数目，然后用对两个
数字求和的BinaryOperator来归约这个流，如下所示：

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

用更为传统的Java术语来说，这段代码与下面的迭代等价：

```
public static long iterativeSum(long n) {
    long result = 0;
    for (long i = 1L; i <= n; i++) {
        result += i;
    }
    return result;
}
```

这似乎是利用并行处理的好机会，特别是n很大的时候。那怎么入手呢？你要对结果变量进
行同步吗？用多少个线程呢？谁负责生成数呢？谁来做加法呢？用并行流的话，这问题就简单多了！

将顺序流转为并行流

你可以把流转换成并行流，从而让前面的函数归约过程（也就是求和）并行运行——————对顺序流调用parallel方法：

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

在上面的代码中，对流中所有数字求和的过程和以往不同
之处在于Stream在内部分成了几块。因此可以对不同的块独立并行进行归约操作，
最后，同一个归约操作会将各个子流的部分归纳结果合并起来，得到整个原始流的归纳结果。

> 请注意，在现实中，对顺序流调用parallel方法并不意味着流本身有任何实际的变化。它在内部实际上就是设了一个boolean标志，表示你想让调用parallel之后进行的所有操作都并行执行。类似地，你只需要对并行流调用sequential方法就可以把它变成顺序流。
>
> 但最后一次parallel或sequential调用会影响整个流水线。也就是一整个流操作中只会被最后一次的parallel或sequential影响

配合并行流使用的线程池

看看流的parallel方法，你可能会想，并行流用的线程是哪儿来的？有多少个？怎么自定义这个过程呢？

并行流内部使用了默认的ForkJoinPool（分支/合并框架会提到），它默认的
线程数量就是你的处理器数量，这个值是由 Runtime.getRuntime().availableProcessors()得到的。

但是你可以通过系统属性 java.util.concurrent.ForkJoinPool.common.parallelism来改变线程池大小，如下所示：

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

这是一个全局设置，因此它将影响代码中所有的并行流。反过来说，目前还无法专为–个并行流指定这个值。
一般而言，最好让ForkJoinPool的大小等于处理器数量，除非你有很好的理由，否则强烈建议不要修改。

现在你有三个方法，用三种不同的方式（迭代式、顺序归纳和并行归纳）做完全相同的操作，
让我们看看谁最快吧！

测量流性能

我们声称并行求和方法应该比顺序和迭代方法性能好。然而在软件工程上，靠猜绝对不是什么好办法！特别是在优化性能时，你应该始终遵循三个黄金规则：测量，测量，再测量。

测量方法如下所示：

```
// 测量队前n个自然数求和的性能
public long measureSumPerf(Function<Long, Long> adder, long n) {
    long fastest = Long.MAX_VALUE;
    for (int i = 0; i < 10; i++) {

        long start = System.nanoTime();
        long sum = adder.apply(n);
        long duration = (System.nanoTime() - start) / 1_000_000;

        System.out.println("Result: " + sum);
        if (duration < fastest)
                fastest = duration;
    }
    return fastest;
}
```

这个方法接受一个函数和一个long作为参数。它会对传给方法的long应用函数10次，记录每次执行的时间（以毫秒为单位），并返回最短的一次执行时间

贴一下ParallelStreams类中三个方法的源码:

```
// 顺序归约版本
public static long sequentialSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
                .limit(n)
                .reduce(0L, Long::sum);
}

// 迭代版本
public static long iterativeSum(long n) {
    long result = 0;
    for (long i = 1L; i <= n; i++) {
        result += i;
    }
    return result;
}

// 并行归约版本
public static long parallelSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
                .limit(n)
```

```
                .parallel()
                .reduce(0L, Long::sum);
}
```

```
// 假设方法都在ParallelStreams中
System.out.println("Sequential sum done in:" +
 measureSumPerf(ParallelStreams::sequentialSum, 10_000_000) + " msecs");
```

请注意，我们对这个结果应持保留态度。影响执行时间的因素有很多，比如你的电脑支持多少个内核。你可以在自己的机器上r一下这些代码。我们在一台四核英特尔i7 2.3 GHz的MacBook Pro上运行它，输出是这样的：

```
Sequential sum done in: 97 msecs
```

用传统for循环的迭代版本执行起来应该会快很多，因为它更为底层，更重要的是不需要对原始类型做任何装箱或拆箱操作。如果你试着测量它的性能，

```
System.out.println("Iterative sum done in:" +
 measureSumPerf(ParallelStreams::iterativeSum, 10_000_000) + " msecs");
```

将得到:

```
Iterative sum done in: 2 msecs
```

现在我们来对函数的并行版本做测试：

```
System.out.println("Parallel sum done in: " +
 measureSumPerf(ParallelStreams::parallelSum, 10_000_000) + " msecs" );
```

看看会出现什么情况:

```
Parallel sum done in: 164 msecs
```

这相当令人失望，求和方法的并行版本比顺序版本要慢很多。你如何解释这个意外的结果呢？这里实际上有两个问题：

- iterate生成的是装箱的对象，必须拆箱成数字才能求和；
- 我们很难把iterate分成多个独立块来并行执行。

第二个问题更有意思一点，因为你必须意识到某些流操作比其他操作更容易并行化。具体来说，iterate很难分成能够独立执行的小块，因为每次应用这个函数都要依赖前一次应用的结果，

这意味着，在这个特定情况下，整个进程不是像前面所说的那样进行的；整张数字列表在归纳过程开始时没有准备好。
因而无法有效的把流划分为小块来并行处理，把流标记成并行，反而给顺序处理增加了开销。它还要把每次求和操作分到一个不同的线程上

这就说明了并行编程可能很复杂，有时候甚至有点违反直觉。如果用得不对（比如采用了一个不易并行化的操作，如iterate），它甚至可能让程序的整体性能更差，所以在调用那个看似神奇的parallel操作时，了解背后到底发生了什么是很有必要的。

使用更有针对性的办法

那到底要怎么利用多核处理器，用流来高效地并行求和呢?

我们在前面讨论了一个叫LongStream.rangeClosed的方法。这个方法与iterate相比有两个优点。

- LongStream.rangeClosed直接产生原始类型的long数字，没有装箱拆箱的风险
- LongStream.rangeClosed会生成数字范围，很容易划分为独立的小块。。例如，范围1~20可分为1~5、6~10、11~15和16~20。

让我们先看一下它用于顺序流时的性能如何，看看装箱和拆箱到底要不要紧：

```
public static long rangedSum(long n) {
    return LongStream.rangeClosed(1, n)
                .reduce(0L, Long::sum);
}
```

这一次的输出是：

```
Ranged sum done in: 17 msecs
```

这个数值流比前面那个用iterate工厂方法生成数字的顺序执行版本要快得多，因为数值流避免了那些没必要的自动装箱和拆箱操作。由此可见，选择适当的数据结构往往比并行化算法更重要。但要是对这个新版本应用并行流呢?

```
public static long parallelRangedSum(long n) {
    return LongStream.rangeClosed(1, n)
                    .parallel()
                    .reduce(0L, Long::sum);
}
```

你会得到：

```
Parallel range sum done in: 1 msecs
```

终于，我们得到了一个比顺序执行更快的并行归纳，因为这一次归纳操作可以正确执行了。这也表明，使用正确的数据结构然后使其并行工作能够保证最佳的性能。

尽管如此，请记住，并行化并不是没有代价的。并行化过程本身需要对流做递归划分，把每个子流的归纳操作分配到不同的线程，然后把这些操作的结果合并成一个值。但在多个内核之间移动数据的代价也可能比你想的要大，所以很重要的一点是要保证在内核中并行执行工作的时间比在内核之间传输数据的时间长。总而言之，很多情况下不可能或不方便并行化。然而，在使用并行Stream加速代码之前，你必须确保用得对；如果结果错了，算得快就毫无意义了。让我们来看一个常见的陷阱。

避免共享可变状态

错用并行流而产生错误的首要原因，就是使用的算法改变了某些共享状态。下面是另一种实现对前n个自然数求和的方法，但这会改变一个共享累加器：

```
public static long sideEffectSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).forEach(accumulator::add);
    return accumulator.total;
}
public class Accumulator {
    public long total = 0;
    public void add(long value) { total += value; }
}
```

这种代码有什么问题呢？不幸的是，它真的无可救药，因为它在本质上就是顺序的。每次访问total都会出现数据竞争。如果你试图用同步来修复，那就完全失去并行的意义了。为了说明这一点，让我们试着把Stream变成并行的：

```
public static long sideEffectParallelSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).parallel().forEach(accumulator::add);
    return accumulator.total;
}
```

用测试框架来执行这个方法，并打印每次执行的结果：

```
System.out.println("SideEffect parallel sum done in: " +
    measurePerf(ParallelStreams::sideEffectParallelSum, 10_000_000L) +" msecs" );
```

你可能会得到类似于下面这种输出：

```
Result: 5959989000692
Result: 7425264100768
Result: 6827235020033
Result: 7192970417739
Result: 6714157975331
Result: 7497810541907
Result: 6435348440385
Result: 6999349840672
Result: 7435914379978
Result: 7715125932481
SideEffect parallel sum done in: 49 msecs
```

这回方法的性能无关紧要了，要紧的是每次执行都会返回不同的结果，都离正确值50000005000000差很远。这是由于多个线程在同时访问累加器，执行total += value，而这一句虽然看似简单，却不是一个原子操作。

问题的根源在于，forEach中调用的方法有副作用，它会改变多个线程共享的对象的可变状态。要是你想用并行Stream又不想引发类似的意外，就必须避免这种情况。

高效使用并行流

一般而言，想给出任何关于什么时候该用并行流的定量建议都是不可能也毫无意义的，因为任何类似于“仅当至少有一千个（或一百万个或随便什么数字）元素的时候才用并行流）”的建议对于某台特定机器上的某个特定操作可能是对的，但在略有差异的另一种情况下可能就是大错特错。

尽管如此，我们至少可以提出一些定性意见，帮你决定某个特定情况下是否有必要使用并行流。

- 如果有疑问，测量。把顺序流转成并行流轻而易举，但却不一定是好事。并行流并不总是比顺序流快。此外，并行流有时候会和你的直觉不一致，所以在考虑选择顺序流还是并行流时，第一个也是最重要的建议就是用适当的基准来检查其性能。
- 留意装箱。自动装箱和拆箱操作会大大降低性能。Java 8中有原始类型流（IntStream、LongStream、DoubleStream）来避免这种操作，但i有可能都应该用这些流。
- 有些操作本身在并行流上的性能就比顺序流差。特别是limit和findFirst等依赖于元素顺序的操作，它们在并行流上执行的代价非常大。例如，findAny会比findFirst性能好，因为它不一定要按顺序来执行。你总是可以调用unordered方法来把有序流变成无序流。那么，如果你需要流中的n个元素而不是专门要前n个的话，对无序并行流调用limit可能会比单个有序流（比如数据源是一个List）更高效。
- 还要考虑流的操作流水线的总计算成本。设N是要处理的元素的总数，Q是一个元素通过流水线的大致处理成本，则N*Q就是这个对成本的一个Ꮒ略的定性估计。Q值较高就意味着使用并行流时性能好的可能性比较大。
- 对于较小的数据量，选择并行流几乎从来都不是一个好的决定。并行处理少数几个元素

的好处还抵不上并行化造成的额外开销。

- 要考虑流背后的数据结构是否易于分解。例如，ArrayList的拆分效率比LinkedList高得多，因为前者用不着遍历就可以平均拆分，而后者则必须遍历。另外，用range工厂方法创建的原始类型流也可以快速分解。最后，你将在7.3节中学到，你可以自己实现Spliterator来完全掌控分解过程。
- 流自身的特点，以及流水线中的中间操作修改流的方式，都可能会改变分解过程的性能。例如，一个SIZED流可以分成大小相等的两部分，这样每个部分都可以比较高效地并行处理，但筛选操作可能丢弃的元素个数却无法预测，导致流本身的大小未知。
- 还要考虑终端操作中合并步骤的代价是大是小（例如Collector中的combiner方法）。如果这一步代价很大，那么组合每个子流产生的部分结果所出的代价就可能会超出通过并行流得到的性能提升。

下表按照可分解性总结了一些流数据源适不适于并行。

源	可分解性
ArrayList	极佳
LinkedList	差
IntStream.range	极佳
Stream.iterate	差
HashSet	好
TreeSet	好

最后，还要强调一下并行流背后使用的基础架构是Java 7中引入的分支/合并框架。并行汇总的示例证明了要想正确使用并行流，了解它的内部原理至关重要，所以我们稍后会仔细研究分支/合并框架。

分支/合并框架

分支/合并框架的目的是以递归的方式将可以并行的任务拆分成更小的任务，然后将每个子任务的结果合并起来生成整体结果。它是ExecutorService接口的一个实现，它把子任务分配给线程池（称为ForkJoinPool）中的工作线程。首先来看看如何定义任务和子任务。

使用RecursiveTask

要把任务提交到这个池，必须创建RecursiveTask的一个子类，其中R是并行化任务（以及所有子任务）产生的结果类型，或者如果任务不返回结果，则是RecursiveAction类型（当然它可能会更新其他非局部机构）。要定义RecursiveTask，只需实现它唯一的抽象方法compute：

```
protected abstract R compute();
```

这个方法同时定义了将任务划分成子任务的逻辑，以及无法再划分或不方便再划分时，生成单个子任务结果的逻辑。正由于此，这个方法的实现类似于下面的伪代码：

```
if (任务足够小或不可分) {
    顺序计算该任务
} else {
    将任务分成两个子任务
    递归调用本方法，拆分每个子任务，等待所有子任务完成
    合并每个子任务的结果
}
```

下面是使用分支/合并框架实现并发进行累积和计算的代码:

```
public class ForkJoinSumCalculator extends RecursiveTask<Long> {

    private final long[] numbers;
    private final int start;
    private final int end;

    private static final long THRESHOLD = 10_000;

    private ForkJoinSumCalculator(long[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
    }

    public ForkJoinSumCalculator(long[] numbers) {
        this(numbers,0,numbers.length);
    }

    /**
     * Fork 就是把一个大任务切分为若干子任务并行的执行,
     * Join 就是合并这些子任务的执行结果，最后得到这个大任务的结果
     * @Author: CaoTing
     * @Date: 2019/12/22
     */
    @Override
    protected Long compute() {
        int length = end - start;
        if (length <= THRESHOLD) {
            return computeSequentially();
        }

        ForkJoinSumCalculator leftTask =
```

```
                new ForkJoinSumCalculator(numbers, start, start + length/2);

        // 使用另一个ForkJoinPool线程异步执行新创建的子任务
        leftTask.fork();

        ForkJoinSumCalculator rightTask =
                new ForkJoinSumCalculator(numbers, start + length/2, end);

        // 继续在这个线程递归调用compute方法，可能会在上一步fork时再次分支
        Long rightResult = rightTask.compute();

        // 读取另一半子任务的结果，如果未完成会继续等待
        Long leftResult = leftTask.join();
        return leftResult+rightResult;
    }

    /**
     * 在子任务不再分裂时调用的计算方法
     * @Author: CaoTing
     * @Date: 2019/12/22
     */
    private Long computeSequentially() {
        long sum = 0;
        for (int i = start; i< end; i++) {
            sum += numbers[i];
        }
        return sum;
    }
}
```

使用分支/合并框架的最好做法

虽然分支/合并框架还算简单易用，不幸的是它也很容易被误用。以下是几个有效使用它的最佳做法。

- 对一个任务调用join方法会阻塞调用方，直到该任务做出结果。因此，有必要在两个子任务的计算都开始之后再调用它。否则，你得到的版本会比原始的顺序算法更慢更复杂
- 不应该在RecursiveTask内部使用ForkJoinPool的invoke方法。相反，你应该始终直接调用compute或fork方法，只有顺序代码才应该用invoke来启动并行计算。
- 对子任务调用fork方法可以把它排进ForkJoinPool。同时对左边和右边的子任务调用它似乎很自然，但这样做的效率要比直接对其中一个调用compute低。这样做你可以为其中一个子任务重用同一线程，从而避免在线程池中多分配一个任务造成的开销。
- 调试使用分支/合并框架的并行计算可能有点棘手。特别是你平常都在你喜欢的IDE里面看栈跟踪（stack trace）来找问题，但放在分支/合并计算上就不行了，因为调用compute

的线程并不是概念上的调用方，后者是调用fork的那个。
- 和并行流一样，你不应理所当然地认为在多核处理器上使用分支/合并框架就比顺序计算快。

对于分支/合并拆分策略还有最后一点补充：你必须选择一个标准，来决定任务是要进一步划分还是已小到可以顺序求值。

工作窃取

在ForkJoinSumCalculator的例子中，我们决定在要求和的数组中最多包含10 000个项目时就不再创建子任务了。这个选择是很随意的，但大多数情况下也很难找到一个好的启发式方法来确定它，只能试几个不同的值来优化它。

但分出大量的小任务一般来说都是一个好的选择。这是因为，理想情况下，划分并行任务时，应该让每个任务都用完全相同的时间完成，让所有的CPU内核都同样繁忙。但是实际中，每个子任务所花的时间可能天差地别，要么是因为划分策略效率低，要么是有不可预知的原因，比如磁盘访问慢，或是需要和外部服务协调执行。

分支/合并框架工程用一种称为工作窃取（work stealing）的技术来解决这个问题。在实际应用中，这意味着这些任务差不多被平均分配到ForkJoinPool中的所有线程上。每个线程都为分配给它的任务保存一个双向链式队列，每完成一个任务，就会从队列头上取出下一个任务开始执行。

基于前面所述的原因，某个线程可能早早完成了分配给它的所有任务，也就是它的队列已经空了，而其他的线程还很忙。这时，这个线程并没有闲下来，而是随机选了一个别的线程，从队列的尾巴上“偷走”一个任务。这个过程一直持续下去，直到所有的任务都执行完毕，所有的队列都清空。这就是为什么要划分成许多小任务而不是少数几个大任务，这有助于更好地在工作线程之间平衡负载。

一般来说，这种工作窃取算法用于在池中的工作线程之间重新分配和平衡任务。

Spliterator

Spliterator是Java 8中加入的另一个新接口；这个名字代表“可分迭代器”（splitable iterator）。和Iterator一样，Spliterator也用于遍历数据源中的元素，但它是为了并行执行而设计的。

Spliterator接口

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

- T是Spliterator遍历的元素的类型。
- tryAdvance方法的行为类似于普通的Iterator，因为它会按顺序一个一个使用Spliterator中的元素，并且如果还有其他元素要遍
  历就返回true。
- trySplit是专为Spliterator接口设计的，因为它可以把一些元素划出去分
  给第二个Spliterator（由该方法返回），让它们两个并行处理。
- estimateSize方法可以估计还剩下多少元素要遍历。

拆分过程

将Stream拆分成多个部分的算法是一个递归过程。第一步是对第一个Spliterator调用trySplit，生成第二个Spliterator。第二步对这两个Spliterator调用trysplit，这样总共就有了四个Spliterator。这个框架不断对Spliterator调用trySplit直到它返回null，表明它处理的数据结构不能再分K，如第三步所示。最后，这个递ॅ٧分过程到第四步就终止了，这时所有的Spliterator在调用trySplit时都返回了null。

这个拆分过程也受Spliterator本身的特性影响，而特性是通过characteristics方法声明的。

Spliterator的特性

Spliterator接口声明的最后一个抽象方法是characteristics，它将返回一个int，代表Spliterator本身特性集的编码。下表总结了这些特性

表7-2 Spliterator的特性

特　性	含　义
ORDERED	元素有既定的顺序（例如List），因此Spliterator在遍历和划分时也会遵循这一顺序
DISTINCT	对于任意一对遍历过的元素x和y，x.equals(y)返回false
SORTED	遍历的元素按照一个预定义的顺序排序
SIZED	该Spliterator由一个已知大小的源建立（例如Set），因此estimatedSize()返回的是准确值
NONNULL	保证遍历的元素不会为null
IMMUTABLE	Spliterator的数据源不能修改。这意味着在遍历时不能添加、删除或修改任何元素
CONCURRENT	该Spliterator的数据源可以被其他线程同时修改而无需同步
SUBSIZED	该Spliterator和所有从它拆分出来的Spliterator都是SIZED

实现你自己的Spliterator

让我们来看一个可能需要你自己实现Spliterator的实际例子。我们要开发一个简单的方法来数数一个String中的单词数。这个方法的一个迭代版本可以写成下面的样子：

```
public int countWordsIteratively(String s) {

    int counter = 0;
    boolean lastSpace = true;

    for (char c : s.toCharArray()) {
```

```
        if (Character.isWhitespace(c)) {
            lastSpace = true;
        } else {
            if (lastSpace) counter++;
                lastSpace = false;
        }
    }
    return counter;
}
```

现在调用它：

```
final String SENTENCE =
    " Nel mezzo del cammin di nostra vita " +
    "mi ritrovai in una selva oscura" +
    " ché la dritta via era smarrita ";

System.out.println("Found " + countWordsIteratively(SENTENCE) + " words");
```

我在句子里添加了一些额外的随机空格，以演示这个迭代实现即使在两个词之间
存在多个空格时也能正常工作，这段代码将打印以下内容：

```
Found 19 words
```

理想情况下，你会想要用更为函数式的风格来实现它，因为就像我们前面说过的，这样你就
可以用并行Stream来并行化这个过程，而无需显式地处理线程和同步问题。

1. 以函数式风格重写单词计数器

首先你需要把String转换成一个流。但是原始类型的流仅限于int、long和double，
所以你只能用Stream：

```
Stream<Character> stream = IntStream.range(0, SENTENCE.length())
                                    .mapToObj(SENTENCE::charAt);
```

-->