转载：<http://www.cnblogs.com/Stay-Hungry-Stay-Foolish/p/7435414.html>

# Lambda表达式

## 行为参数化

### 引入

在软件工程中，一个众所周知的问题就是：不管你做什么，用户的需求肯定会变。比如说，有一个应用程序是帮助农民了解自己的库存的。这位农民可能想有一个查找库存中所有的绿苹果的功能。但是到了第二天，他可能告诉你，其实还想找出所有重量超过150g的苹果。又过了两天，农民又跑过来补充道，要是我可以找出所有既是绿色，重量也超过150g的苹果，那就太好了。你要如何应对这样不断变化的需求呢？

行为参数化就是可以帮助你处理频繁变更的需求的一种软件开发模式。总之，它意味着拿出一个代码块，把它准备好却不去执行它。这个代码块以后可以被你的程序的其他部分调用，这就表示你可以推迟这个代码块的执行。例如，你可以将代码块作为参数传给一个方法，在这个方法里面再去执行它。这样，这个方法的行为就基于那个代码块被参数化了。

编写能够应对变化的需求的代码并不容易。让我们看看一个例子，我们会逐步地改进这个例子，以展示一些让代码更加灵活的最佳做法。就农场库存程序而言，你必须实现一个从列表中筛选绿苹果的功能。

第一个解决方案可能是下面这样的：

private List<Apple> filterGreenApple(List<Apple> inventory) {

List<Apple> result = new ArrayList<>(); // 累积的苹果列表

for (Apple apple : inventory) {

if ("green".equals(apple.getColor())) { //仅仅筛选出绿苹果

result.add(apple);

}

}

return result;

}

这里筛选条件是筛选绿苹果。但是现在农民改主意了，他还要筛选红苹果，你该怎么做呢？简单的方法就是复制这段代码，将名字改为filterRedApple，然后更改if条件来匹配红苹果。然而，要是农民想要筛选多种颜色：浅绿色、暗红色、黄色等等，这种方法就应付不了。一个良好的原则是在编写类似的代码之后，尝试将其抽象化。

一种做法是给方法加一个参数，把颜色变为参数，这样就能灵活的适应变化了：

private List<Apple> filterAppleByColor(List<Apple> inventory, String color) {

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (color.equals(apple.getColor())) {

result.add(apple);

}

}

return result;

}

现在，只要像下面这样调用方法，农民朋友就会满意了：

List<Apple> greenApples = filterAppleByColor(inventory, "green");

List<Apple> redApples = filterAppleByColor(inventory, "red");

太简单了对吧？那我们在把例子弄得更加复杂一点儿。这位农民又跑过来和你说，要是能够区分轻的苹果和重的苹果就太好了，重的苹果一般是重量大于150g。于是你写了下面的方法，用一个参数来应对不同的重量：

private List<Apple> filterAppleByWeight(List<Apple> inventory, int weight) {

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (apple.getWeight() > weight) {

result.add(apple);

}

}

return result;

}

解决方法不错，但是复制大部分的代码来实现遍历库存，并对每一个苹果应用筛选条件。这让人有点失望，因为它打破了DRY(Don't Repeat Yourself，不要重复自己)的软件工程原则。如果你想要改变筛选方式来提升性能，那就得修改所有方法的实现，而不是只改一个。从工程的工作量角度来看，这个代价太大了。

### 行为参数化

在上面已经看到了，你需要一种比添加很多参数更好的方法来应对变化的需求。让我们后退一步来看看更高层次的抽象。一种可能的解决方案是对你的选择标准进行建模。让我们定义一个接口来对选择标准进行建模：

public interface ApplePredicate {

boolean test(Apple apple);

}

现在可以使用ApplePredicate的多个实例代表不同的选择标准了，比如：

public class AooleHeavyWeightPericate implements ApplePredicate { //选出重的苹果

@Override

public boolean test(Apple apple) {

return apple.getWeight() > 150;

}

}

public class AppleGreenColorPerdicate implements ApplePredicate { //选择绿苹果

@Override

public boolean test(Apple apple) {

return "green".equals(apple.getColor());

}

}

使用ApplePredicate改过之后，filter方法看起来是这样的：

private List<Apple> filterApple(List<Apple> inventory, ApplePredicate p){

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (p.test(apple)) {

result.add(apple);

}

}

return result;

}

我们都知道，人们不愿意使用那些很麻烦的功能或者概念。目前，当要把新的行为传递给filterApples方法的时候，你不得不声明好几个实现ApplePredicate接口的类，然后去实例化好几个只会提到一次的ApplePredicate对象。下面的程序总结了你目前看到的一切。这个真是非常的啰嗦，很费时间！

public class AppleHeavyWeightPericate implements ApplePredicate { //选出重的苹果

@Override

public boolean test(Apple apple) {

return apple.getWeight() > 150;

}

}

public class AppleGreenColorPerdicate implements ApplePredicate {//选择绿苹果

@Override

public boolean test(Apple apple) {

return "green".equals(apple.getColor());

}

}

public class Demo {

public static void main(String[] args) {

List<Apple> inventory = Arrays.asList(

new Apple("green", 80),

new Apple("green", 155),

new Apple("red", 120));

List<Apple> greenApples = filterApple(inventory,

new AppleGreenColorPerdicate());

List<Apple> heavyWeightApples= filterApple(inventory,

new AppleHeavyWeightPericate());

}

private static List<Apple> filterApple(List<Apple> inventory, ApplePredicate p) {

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (p.test(apple)) {

result.add(apple);

}

}

return result;

}

}

费这么大劲儿真没必要，能不能做的更好呢？

### 匿名类

 Java有一个机制称为匿名类，它可以让你同时声明和实例化一个类，这可以帮助你进一步改善代码，让代码变得更加的简洁。

匿名类和你熟悉的Java局部类(块中定义的类)差不多，但匿名类没有名字。它允许你同时声明并且实例化一个类。换句话说，它允许你随用随建。

下面的代码展示了如何通过创建一个匿名类实现ApplePredicate的对象，重写筛选的例子：

List<Apple> redApples = filterApple(inventory, new ApplePredicate() {

@Override

public boolean test(Apple apple) {

return "red".equals(apple.getColor());

}

});

但匿名类还是不够好。它往往显得非常的笨重，因为占用了很多空间。比如：

List<Apple> redApples = filterApple(inventory, new ApplePredicate() {

@Override

public boolean test(Apple apple) {

return "red".equals(apple.getColor());

}

});

List<Apple> greenApples = filterApple(inventory, new ApplePredicate() {

@Override

public boolean test(Apple apple) {

return "green".equals(apple.getColor());

}

});

上面代码中红色部分都是重复的。

整体来说，啰嗦就不好，它让人不愿意使用语言的某种功能，因为编写和维护啰嗦的代码需要很多时间，而且代码也不易读。好的代码应该是一目了然的。即使使用匿名类处理在某种程度上盖过了为一个接口声明好几个实体类的啰嗦问题，但是它仍然不能令人满意。

### Lambda表达式

上面的代码在Java8里可以使用Lambda表达式重写为下面的样子：

List<Apple> result = filterApple(inventory,

(Apple apple) ->"red".equals(apple.getColor()));

 不得不承认这代码看上去比先前干净很多。这个很好，因为它看起来更像问题描述本身了。我们现在已经解决了啰嗦的问题了。

 在通向抽象的路上，我们还可以更进一步。目前，filterApples方法还只适用于Apple。你还可以将List类型抽象化，从而超越你眼前要处理的问题：

public interface ApplePredicate<T> {

boolean test(T t);

}

public class Demo {

public static void main(String[] args) {

List<Apple> inventory = Arrays.asList(

new Apple("green", 80),

new Apple("green", 155),

new Apple("red", 120));

List<Apple> result = filterApple(inventory,

(Apple apple) ->"red".equals(apple.getColor()));

for(Apple apple : result){

System.out.println(apple);

}

}

private static <T>List<T> filterApple(List<T> inventory, ApplePredicate<T> p) {

List<T> result = new ArrayList<>();

for (T t : inventory) {

if (p.test(t)) {

result.add(t);

}

}

return result;

}

}

现在你可以把filter方法用在了香蕉、橘子、Integer或是String的列表上了。酷不酷？你现在在灵活性和间接性之间找到了最佳的平衡点，这在Java8之前是不可能的！

### 真实的例子

 你现在已经看到了，行为参数化是一个很有用的模式，它能够轻松的适应不断变化的需求。这种模式可以把一个行为(一段代码)封装起来，并通过传递和使用创建的行为将方法的行为参数化。这种做法类似于策略设计模式。

1、使用Comparator来排序

 对集合进行排序是一个常见的编程任务。比如，你的那位农民朋友想要根据苹果的重量对库存的进行排序，或者他可能改变了主意，希望你根据颜色对苹果进行排序，听起来怎么有点耳熟呢？是的，你需要一种方法来表示和使用不同的排序行为，来轻松的适应变化的需求。

在Java8中，List自带一个sort方法(你也可以使用Collections.sort)。sort的行为可以用Java.util.Comparator对象来进行参数化，它的接口如下：

public interface Comparator<T> {

int compare(T o1, T o2);

}

因此，你可以随时创建Comparator的实现，用sort方法表现出不同的行为。比如，你可以使用匿名类，按照重量升序对库存进行排序：

inventory.sort(new Comparator<Apple>() {

@Override

public int compare(Apple o1, Apple o2) {

return o1.compareTo(o2);

}

});

用Lambda表达式的话，看起来就是这样：

inventory.sort((Apple o1, Apple o2) ->o1.compareTo(o2));

2、使用Runnable执行代码块

线程就像是轻量级的进程:它们自己执行一个代码块。但是怎么才能告诉线程要执行那块代码呢？多个线程可能会运行不同的代码。我们需要一种方式来代表稍后执行的一段代码。在Java里面，你可以使用Runnable接口表示一个要执行的代码块。请注意，代码不会返回任何结果(即void):

你可以像下面这样，使用这个接口创建执行不同行为的线程：

Thread t = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("hello world");

}

});

使用Lambda表达式，看起来是这样的：

Thread t = new Thread(()->System.out.println("hello world"));

## 语法

使用匿名类来表示不同的行为并不令人满意:代码十分的啰嗦，这会影响程序员在实践中使用行为参数化的积极性。在这里，你会认识Java8中解决这个问题的工具Lambda表达式，它可以让你很简洁地表示一个行为或者传递代码。

你可以把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，可能还有可以抛出的异常列表。Lambda表达式的基本语法是：

(parameters) -> expression

或者：

(parameters) -> {expression;}

注意，带括号与不带括号有一定的区别：不带括号只能是一行语句，并且不能带return，末尾不能加分号；带括号可以是一行也可以是多行，但是如果函数有返回值的话，在末尾必须return，而且每一句后面必须带分号。例如：() -> "hello world"等价于 () -> { return "hello world";}

## 函数式接口

我们在之前实现filter方法中是这样使用Lambda的：

List<Apple> result = filterApple(inventory, (Apple apple) ->"red".equals(apple.getColor()));

那么，在哪里可以使用Lambda呢？你可以在函数式接口上使用Lambda表达式。在上面的代码中，你可以把Lambda表达式作为第二个参数传递给filter方法，这里需要一个Predicate<T>接口，而这个接口就是一个函数式接口，因为Predicate接口仅仅定义了一个抽象方法：

public interface ApplePredicate<T> {

boolean test(T t);

}

简而言之，函数式接口就是只定义了一个抽象方法的接口。注意，Java8中的接口还可以默认方法和静态方法，但只要接口只定义了一个抽象方法，该接口就仍然是一个函数式接口。

用函数式接口可以干什么呢？Lamba表达式允许你可以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。你用匿名内部类也可以完成同样的事情，只不过比较笨拙：需要提供一个实现，然后再直接内联将其实例化。如下；

// 使用Lambda

Runnable r1 = () -> System.out.println("hello world");

// 使用匿名类

Runnable r2 = new Runnable() {

@Override

public void run() {

System.out.println("Hello World");

}

};

## 函数描述符

函数式接口的抽象方法的签名基本上就是Lambda表达式的签名，我们把这种抽象方法叫做函数描述符，例如：() -> void代表参数列表为空且返回void的函数，(Apple, Apple) -> int代表接收两个Appe作为参数并且返回int的函数。

你可能在想，Lambda表达式时如何做类型检查的，这个之后再细说。现在，你只需要知道Lambda表达式可以被赋值给一个变量，或者传递给一个接收函数式接口作为参数的方法就行，当然，这个Lambda表达式的签名要和函数式接口的抽象方法的签名一致，例如：

public void process(Runnable r) {

r.run();

}

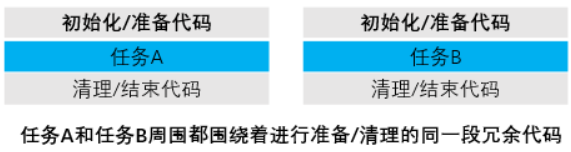
process(() -> System.out.println("hello world"));

上面的代码打印的是"hello world"，Lambda表达式：() -> System.out.println("hello world")不接收参数并且返回void，这个恰恰就是Runnable接口中run()方法的签名。

 如果你去看看新的Java API，会发现函数式接口带有@FunctionalInterface的注解。这个注解用于表示该接口会设计成一个函数式接口，如果使用了@FunctionalInterface定义了一个接口，而它却不是函数式接口的话，编译器将返回一个提示原因的错误，例如错误信息可能是:Multiple non-overriding abstract methods found in interface Foo,表明存在多个抽象方法。需要注意的是：@FunctionalInterface不是必需的，但是对于为此设计的接口而言，使用它是比较好的做法，它就好像是@Override注解表示方法被重写了。

## 环绕执行模式

 让我们通过一个例子，看看在实践中如何利用Lambda和行为参数化来让代码更加灵活，更加简洁。资源处理(例如处理文件或者数据库)时，一个常见的模式就是打开一个资源，做一些处理，然后再关闭资源。这个打开资源和清理资源阶段总是很类似，并且会围绕着执行处理的代码，这就是所谓的环绕执行(execute around)模式。如下图所示：



例如：在以下代码中，红色部分就是从一个从文件中读取一行所需的模板代码(注意：Java7中，带资源的try语句会自动关闭资源，所以你不需要显示地关闭资源了)：

public static String processFile() throws IOException{

try(BufferedReader read = new BufferedReader(new FileReader("D://data.txt"))){

return read.readLine();

}

}

上面那段代码是有局限性的，你只能读文件的第一行，如果你想要读取文件的前两行，甚至是返回使用最频繁的词，该怎么办呢？在理想的情况下，你要重用打开资源和清理资源的代码，并且告诉processFile()方法对文件执行不同的操作。这听起来是不是很耳熟呢？是的，你需要把processFile的行为参数化。你需要一种方法把行为传递给processFile方法，以便它可以利用BufferedReader执行不同的行为。

（1）创建函数式接口

public interface BufferedReaderProcessor {

String process(BufferedReader b) throws IOException;

}

（2）执行一个行为

任何BufferedReader -> String形式的Lambda都可以可以作为参数来传递，因为它们都符合BufferedReaderProcessor接口中定义的process方法的签名。请记住，Lambda表达式允许你直接内联为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。因此，你可以在processFile主体内，对传递过来的BufferReaderProcessor对象调用process方法执行处理：

public static String processFile(BufferedReaderProcessor b) throws IOException{

try(BufferedReader br = new BufferedReader(new FileReader("D://data.txt"))){

return b.process(br);

}

}

（3）传递Lambda

现在你就可以传递不同的Lambda来重用precessFile方法，并用不同的方式处理文件了。

处理一行：

String result = processFile((BufferedReader p) -> p.readLine());

处理两行：

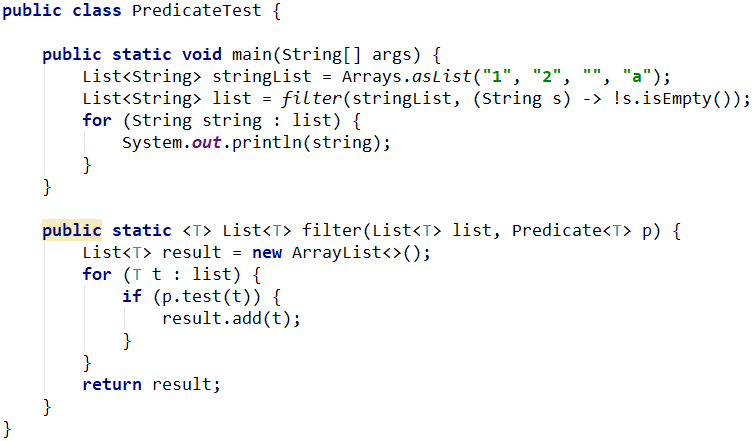
String result = processFile((BufferedReader p) -> p.readLine() + p.readLine());

## 系统中的函数式接口

函数式接口的抽象方法的签名被称为函数描述符。因为函数式接口的抽象方法的签名可以描述Lambda表达式的签名，所以为了应用不同的Lambda表达式，你需要一套能够描述常见函数描述符的函数式接口。Java API中已经提供了几个函数式接口，例如Runnable、Callalbe、Comparable，此外，Java8中的java.util.function包中还引入了几个新的函数式接口，例如Predicate、Consumer、Function等。

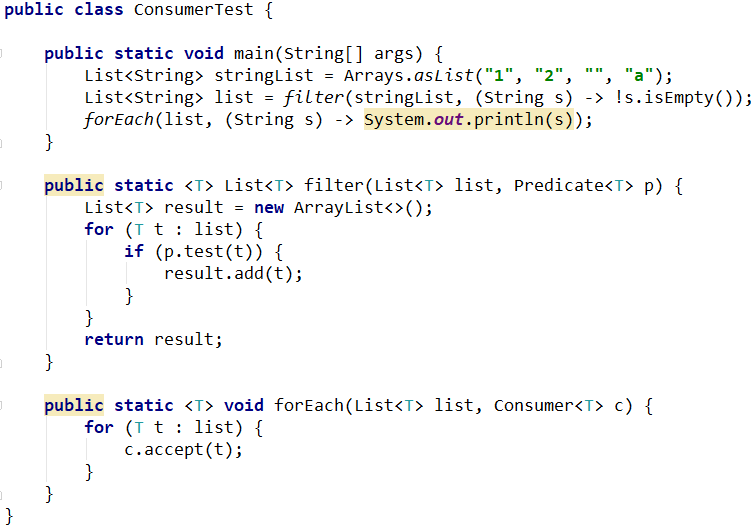
1. Predicate

java.util.function.Predicate<T>接口定义了一个名为test的抽象方法，它接收泛型T对象，并返回一个boolean类型。



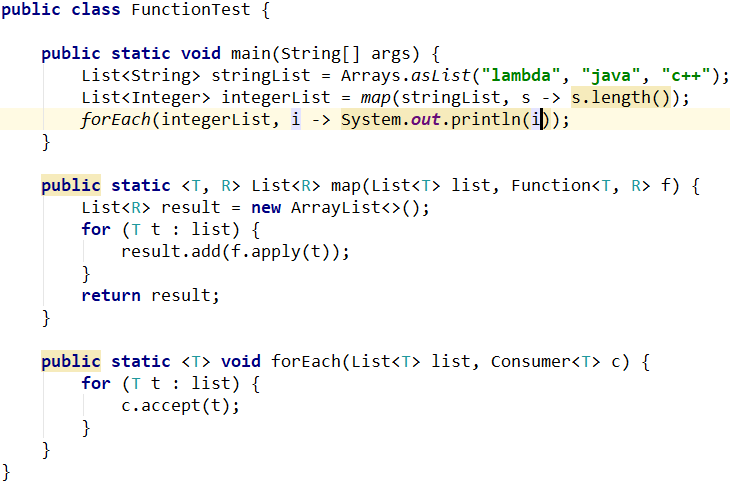
2、Consumer

Java.util.function.Consumer<T>定义了一个名为accept的抽象方法，它接收泛型T对象，并返回void。



3、Function

Java.util.function.Function<T, R>接口定义了一个名为apply的方法，它接收一个泛型T对象，并返回一个泛型R对象。如果你需要定义一个Lambda，将输入对象映射到输出对象，就可以使用这个接口。



4、原始类型特化

我们已经介绍了三个泛型函数式接口：Predicate<T>、Consumer<T>和Function<T, R>，还有一些函数式接口是专为原始类型设计的。

回顾一下：Java类型要么是引用类型，比如Byte、Integer、Object、List等，要么是原始类型，比如int、double、byte、char等。但是泛型只能绑定到引用类型，这是由于泛型内部的实现方式造成的，因此，在Java里有一个将原始类型转换为对应的引用类型的机制，这个机制叫做装箱，而相反的操作，即将引用类型转换为对应的原始类型，就叫做拆箱。另外，Java还有一个自动装箱拆箱机制来帮助程序员执行这个任务，即装箱和拆箱是自动完成的，但是，自动装箱拆箱在性能方面要付出代价：装箱的本质就是把原始类型包裹起来，并保存到堆里，因此装箱后的值需要更多的内存，且需要额外的内存搜索来获取被包裹的原始类型值。

Java8为原始类型提供了专门的函数式接口，以便在输入和输出都是原始类型时避免自动装箱操作。比如下面的代码中，使用intPredicate就避免对值1000进行装箱操作，但如果使用Predicate<Integer>，则会把参数1000装箱到一个Integer对象中：

IntPredicate evenNumbers = (int i) -> i % 2 == 0;

evenNumbers.test(1000);

Predicate<Integer> oddNumbers = (Integer i) -> i % 2== 0;

oddNumbers.test(1000);

一般来说，针对输入参数类型为原始类型的函数式接口的名称都会加上对应的原始类型前缀，比如DoublePredicate、IntConsumer、LongBinaryOperator、IntFunction<R>等。另外，Function接口还有针对输出参数类型为原始类型的函数式接口，比如ToIntFunction<T>、IntToDoubleFucntion等。

下表中总结了Java API中提供的最常用的函数式接口及其函数描述符。

| 函数式接口 | 函数描述符 | 原始类型转换 |
| --- | --- | --- |
| Predicate<T> | T –> boolean | IntPredicate、LongPredicate、DoublePredicate |
| Consumer<T> | T -> void | IntConsumer、LongConsumer、DoubleConsumer |
| Supplier<T> | () -> T | BooleanSupplier、IntSupplier、LongSupplier、DoubleSupplier |
| Function<T, R> | T -> R | IntFunction<R>、LongFunction<R>、DoubleFunction<R>、ToIntFunction<T>、  ToLongFunction<T>、ToDoubleFunction<T>、  IntToLongFunction、IntToDoubleFunction、  LongToDoubleFunction、LongToIntFucntion、  DoubleToIntFunction、DoubleToLongFucntion |
| 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> |
| UnaryOperator<T> | T -> T | IntUnaryOperator、LongUnaryOperator、DoubleUnaryOperator |
| BinaryOperator<T> | (T, T) -> T | IntBinaryOperator、LongBinaryOperator、DoubleBinaryOperator |

下表中总结了使用函数式接口的一些使用案例。

| 案例 | Lambda | 函数式接口 |
| --- | --- | --- |
| 布尔表达式 | (List<String> list) -> list.isEmpty() | Predicate<List<String>> |
| 创建对象 | () -> new Apple() | Supplier<Apple> |
| 消费对象 | (Apple a) -> System.out.println(a.geWeight()) | Consumer<Apple> |
| 过滤对象 | (String s) -> s.length() | Function<String, Integer>、TOIntFunction<String> |
| 合并两个值 | (int a, int b) -> a \* b | IntBinaryOperator |
| 比较两个对象 | (Apple a1, Apple a2) -> a1.compareTo(a2) | Compartor<Apple>、BiFunction<Apple, Apple, Integer>、ToIntBiFunction<Apple, Apple> |

## Lambda表达式异常

如果你需要Lambda表达式抛出异常，有两种做法：

1. 在定义对应的函数式接口时，让其抛出一个检查性异常
2. 把Lambda表达式放在try-catch语句块中

比如，之前我们定义了一个函数式接口BufferedReaderProcessor，它就显式声明了一个IOException：

public interface BufferedReaderProcessor {

String process(BufferedReader br) throws IOException;

}

如果你使用的是Java8中的Function<T, R>，这个函数式接口没有显式声明一个检查新异常，因此，你需要捕获异常：

Function<BufferedReader, String> f = (BufferedReader b) -> {

try {

return b.readLine();

} catch (IOException e) {

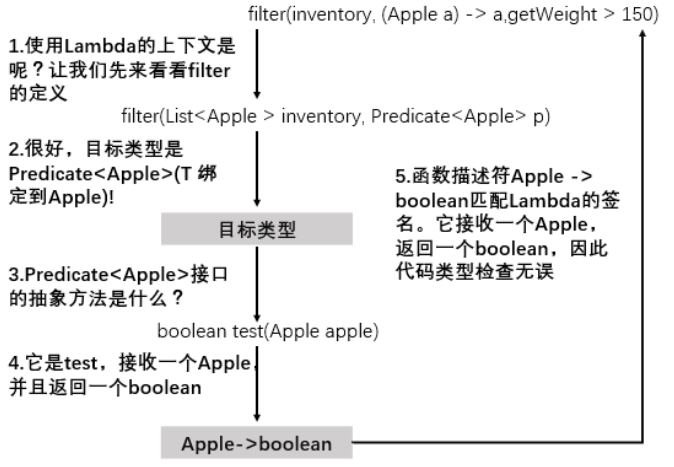
throw new RuntimeException();

}

};

## 类型推导

Lambda的类型是从使用Lambda的上下文推导出来的，上下文中Lambda表达式所需要的类型被称为目标类型。让我们通过一个例子，看看当使用Lmabda表达式时，背后发生了什么。



有了目标类型的概念，同一个Lambda表达式就可以与不同的函数式接口联系起来，只要它们的抽象方法签名能够兼容。比如，前面提到的Callable和PrivilegedAction，这两个接口都代表着什么也不接收并且返回一个泛型T的函数。因此，下面两个赋值是有效的：

Callable<Integer> c = () -> 42;

PrivilegedAction<Integer> p = () -> 42;

这里，第一个赋值的目标类型是Callable<Integer>, 第二个赋值的目标类型是PrivilegedAction<Integer>。

同理，下面三个赋值也是有效的：

Comparator<Apple> c1 = (Apple a1, Apple a2) -> a1.compareTo(a2);

ToIntBiFunction<Apple, Apple> c2 = (Apple a1, Apple a2) -> a1.compareTo(a2);

BiFunction<Apple, Apple, Integer> c3 = (Apple a1, Apple a2) -> a1.compareTo(a2);

注意，如果一个Lambda表达式的函数体是一个语句表达式，它就和一个返回void的函数描述符兼容，例如，以下两行都是合法的，尽管List的add方法返回一个boolean，而不是Consumer所需的void：

//Predicate返回了一个boolean

Predicate<String> p = (String s) -> stringList.add(s);

//Consumer返回了一个void

Consumer<String> b = s -> stringList.add(s);

另外，编译器会从上下文(目标类型)推断出用什么函数式接口来配合Lambda表达式，这就意味着它也可以推断出Lambda的签名，所以我们可以在Lambda语法中就不用显式的给出参数类型，从而提高代码的可读性。也就是说，Java编译器会像下面这样推断Lambda的参数类型

//apple 没有显式的给出类型

List<Apple> result = filterApple(inventory, apple ->"red".equals(apple.getColor()));

Lambda表达式有多个参数，代码可读性的好处就显得更加明显，例如创建一个Comparator对象：

Comparator<Apple> c1 = (Apple a1, Apple a2) -> a1.compareTo(a2);

Comparator<Apple> c2 = (a1, a2) -> a1.compareTo(a2);

## 变量捕获

我们迄今为止所介绍的所有Lambda表达式都只用在方法的参数上，但是Lambda表达式也允许捕获变量，例如下面的Lambda捕获了portNumber变量：

int portNumber = 1337;

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

对于变量捕获，Lambda有一些限制：Lambda可以没有限制的捕获实例变量和静态变量，但是局部变量必须显式的声明为final，或者本质上是final变量，即如果一个局部变量在初始化后从未被修改过，那么它就可以被捕获，换句话说，加上final后也不会导致编译错误的局部变量就是可以被Lambda捕获的变量。例如，下面的变量无法编译，因为portNumer变量被赋值了两次：

int portNumber = 1337;

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

portNumber = 10;

## 方法引用

 方法的引用让你可以重复使用现有的方法定义，并像Lambda一样传递它们。在一些情况下，比起使用Lambda表达式，方法引用似乎更加容易读，感觉也更加自然。下面就是我们借助更新的Java8 API，用方法的引用写的一个排序例子：

// 先前

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

// 之后

Inventory.sort(Comparator.comparing(Apple::getWeight));

方法引用可以被看做是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表达式只是“直接调用这个方法”，那最好还是用名称来调用它，而不是去描述如何调用。事实上，方法引用就是让你根据已有的方法实现来创建Lambda表达式，其工作方式是：当你需要使用方法引用时，将目标引用放在分隔符::前，方法名称放在分隔符后。例如Apple::getWeight就是引用了Apple类中定义的getWeight方法，这是Lambda表达式(Apple a) -> a.getWeight()的快捷写法。注意，不需要括号，因为你没有实际调用这个方法。下表给出了Java8中方法引用的其他例子：

| Lambda表达式 | 等效的方法引用 |
| --- | --- |
| (Apple a) -> a.getWeight() | Apple::getWeight |
| () -> Thread.currentThread.dumpstack() | Thread.currentThread::dumpstack |
| (String str, int i) -> str.substring(i) | String::substring |
| (String s) -> System.out.println(s) | System.out::println |

你可以把方法引用看做是一个针对仅仅涉及单一方法的Lambda语法糖，因为你表达同样的事情时要写的代码更加少了。方法引用和lambda表达式拥有相同的特性，比如它们都需要一个目标类型，并需要被转换为函数式接口的实例，不过我们并不需要为方法引用提供方法体，而是可以直接通过方法名称引用已有方法。

方法引用有很多种，它们的语法如下：

静态方法引用：ClassName:: methName

对象的实例方法引用：instanceRef::methName

对象的超类方法引用：super::methName

类的任意对象上的实例方法引用：ClassName::methName

构造方法引用：Class::new

数组构造方法引用：TypeName[]::new

举几个例子。

1、对一个字符串的List排序，忽略大小写。

List<String> str = Arrays.asList("a", "b", "c", "d");

// str.sort((s1, s2) -> s1.compareToIgnoreCase(s2));

str.sort(String::compareToIgnoreCase);

2、将一个整型字符串转换成整型。

// Function<String, Integer> strToInt = s -> Integer.parseInt(s);

Function<String, Integer> strToInt = Integer::parseInt;

3、判断一个List是否包含某个元素。

// BiPredicate<List<String>, String> contains = (list, element) -> list.contains(element);

BiPredicate<List<String>, String> contains = List::contains;

4、构造无参对象

// Supplier<Apple> supplier = () -> new Apple();

// Supplier<Apple> supplier = new Supplier<Apple>() {

// @Override

// public Apple get() {

// return new Apple();

// }

// }

Supplier<Apple> supplier = Apple::new; // 指向Apple()的构造函数引用

Apple apple = supplier.get();

5、构造带有一个参数的对象

// IntFunction<Apple> f = (weight) -> new Apple(weight);

IntFunction<Apple> f = Apple::new;; // 指向Apple(String color)的构造函数引用

Apple anotherApple = f.apply(70);

6、构造带有两个参数的对象

// BiFunction<Integer, String, Apple> bf = (weight, color) -> new Apple(weight, color);

// 指向Apple(String color, Integer weight)的构造函数引用

BiFunction<Integer, String, Apple> bf = Apple::new;

Apple yetAnotherApple = bf.apply(70, "red");

7、构造带有三个参数的对象

你已经看到了如何将0个、1个、2个参数的构造参数转变为构造函数引用，那要怎么样才能对三个参数的构造函数，比如Color(int r, int g, int b)，使用构造函数引用呢？

 构造函数引用的语法是ClassName::new，那么在这个例子里面就是Color::new，但是你需要与构造函数引用的签名匹配的函数式接口。因为语言本身没有并没有提供这样的函数式接口，所以你可以自己创建一个：

@FunctionalInterface

interface TriFunction<T, U, V, R> {

R apply(T T, U U, V v);

}

现在你就可以像下面这样使用构造函数引用了：

TriFunction<Integer, Integer, Integer, Color> f = Color::new;

## 复合Lambda表达式

Java8的好几个函数式接口都有为方便使用而设计的方法，比如Comparator、Function和Predicate都提供了允许你进行复合的方法。这是什么意思呢？在实践中，这意味着你可以把多个简单的Lambda复合成为复杂的表达式，比如，你可以让两个断言之间做一个or操作从而组合成一个更大的断言。而且，你还可以让一个函数的结果成为另一个函数的输入。你可能会在想，函数式接口中怎么可能会有更多的方法呢？毕竟，这违背了函数式接口的定义。关键在于，我们即将介绍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

  我们前面看到了，你可以使用Comparator.comparing来返回一个Comparator对象，如下所示：

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

（1）逆序

如果你想要对苹果按质量递减排序怎么办？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排序：

inventory.sort(Comparator.comparing(Apple::getWeight).reversed());  
 （2）比较器链

如果发现有两个苹果一样重，哪个苹果应该排在前面呢？你可能需要再提供一个Comparator来进一步定义这一个比较。比如，在按重量比较两个苹果之后，你可能想要按颜色排序。thenComparing方法就是用来做来这个的，它接收一个函数作为参数(就像comparing方法一样)，如果两个对象用第一个Comparator比较之后是一样的，就提供第二个Comparator。你又可以优雅的解决这个问题：

inventory.sort(Comparator.*comparing*(Apple::getWeight).reversed());

### 断言复合

 断言接口包括三个方法：negate、and和or，让你可以重用已有的Predicate来创建更加复杂的断言。比如你可以使用negate方法来返回一个Predicate的非：

Predicate<Apple> redApple = apple -> apple.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又比较重：

Predicate<Apple> redAndHeavyApple = redApple.and(apple -> apple.getWeight() > 150);

你可以进一步组合断言，比如要么是重(150g以上)的红苹果，要么是绿苹果：

Predicate<Apple> redAndHeavyOrGreenApple = redApple

.and(apple -> apple.getWeight() > 150)

.or(apple -> apple.getColor().equals("green"));

注意，and和or是按照在表达式链中的位置，从左往右确定优先级的。因此，a.or(b).and(c)可以看做(a || b) && c。

### 函数复合

最后，你可以把Function接口代表的Lambda表达式复合起来。Function接口为此配了andThen和compose两个默认方法，它们都会返回Function的一个实例。

andThen方法会返回一个函数，它将输入应用到一个给定函数，再将输出应用到另一个函数。比如，假设有一个函数f数字加1(x -> x + 1)，另一个函数给g给数字乘以2，你可以将它们组合成一个函数h，先给数字加1，再给结果乘以2，即g(f(x))。

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);

compose方法与andThen相反，即f(g(x))。

# 流

## 引入

集合对于很多编程任务来说都是非常基本的，几乎对于任何一个Java程序都是不可或缺的，但是集合操作却远远算法上完美。考虑下面两种情况：

1、很多任务逻辑都涉及类似于数据库的操作，比如对想象一下你有一系列菜组成的菜单，你想对菜按照类别进行分类(比如全素菜肴)，或者查找出最贵的菜。大部分数据库都允许你声明式地指定这些操作，比如，以下sql查询语句就可以选出热量较低的菜肴名称：

select name from dishes where calorie < 400

你看，你不需要实现如何根据菜肴的属性进行筛选，你只需要表达你想要什么。这个基本的思路意味着，你用不着担心怎么去显式地实现这些查询语句。

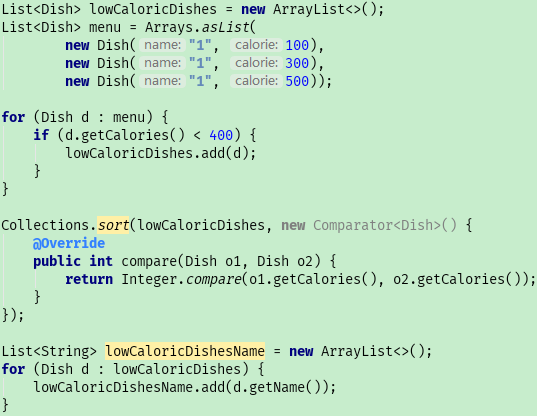
2、要是要处理大量元素又该怎么办呢？为了提供性能，你需要并行处理，并且利用多核架构。但是写并行代码比用迭代器还要复杂，而且调试起来也十分困难。

那Java语言的设计者能做些什么，来帮助程序员节约宝贵的时间呢？你可能已经猜到了，答案就是流。

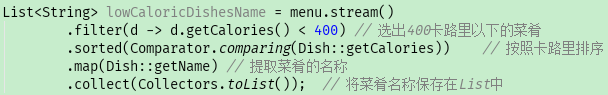
## 流是什么

 流是Java API的新成员，它允许你以声明的方式处理数据集合（通过查询语句来表达，而不是编写一个实现）。此外，流还可以透明地进行并行处理，你无需编写任何多线程代码。

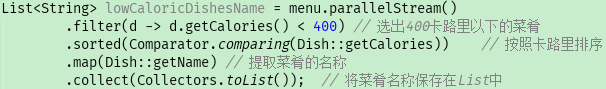
下面我们来简单的看看流的好处。以下两段代码的功能都是返回低热量的菜肴名称，并按照卡路里排序。一个是用Java7写的，另一个是用Java8的流写的。



在这段代码中，你使用了一个“垃圾变量”lowCaloricDishes，它唯一的作用就是作为一次性中间容器。



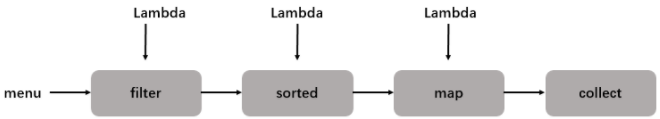
为了利用多核架构并行执行这段代码，你只需要把stream换成parallelStream()：



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：

1、代码是以声明式方法来写的，说明了想要完成什么(筛选热量低的菜肴)，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)。你在之前也看到了，这种方法加上行为参数化让你可以轻松应对变化的需求：你很容易再创建一个代码版本，利用Lamba表达式来筛选高卡路里的菜肴，而不是去复制粘贴代码。

2、你可以把几个基础操作筛选链接起来，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，如下图所示)，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。



因为filter、sorted、map和collect等操作是与具体线程模型无关的高层次构件，所以它们的内部实现可以是单线程的，也可能透明地充分利用你的多级程架构。在实践中，这意味着你用不着为了让某些数据处理任务并行而去操心线程和锁，Stream API都替你想好了。

## 流简介

 要讨论流，我们先来谈谈集合，这是最容易上手的方法。Java8中的集合支持一个新的stream方法，它会返回一个流(接口定义在java.util.stream.Stream里)。你在后面会看到，还有很多的方式来可以得到流，比如利用数值范围或从I/O资源生成的流元素。

  那么，流到底是什么呢？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"。让我们一步步剖析这个定义。

  1、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，但是流的目的在于表达计算，比如你前面简单的filter、sorted和map。集合讲的是数据，流讲的是计算。

 2、源：流会使用一个提供数据的源，如集合、数组或者输入/输出资源。请注意，从有序集合生成流时会保留原有的顺序。

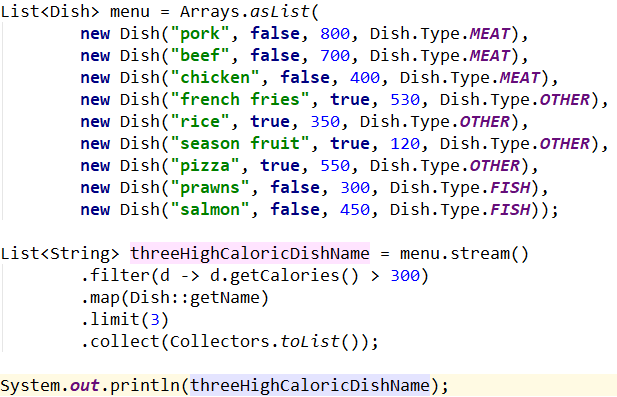
3、数据处理操作：流所支持的数据处理功能类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、sort等等。流操作可以顺序执行，也可以并行执行。

此外，流操作有两个重要的特点。

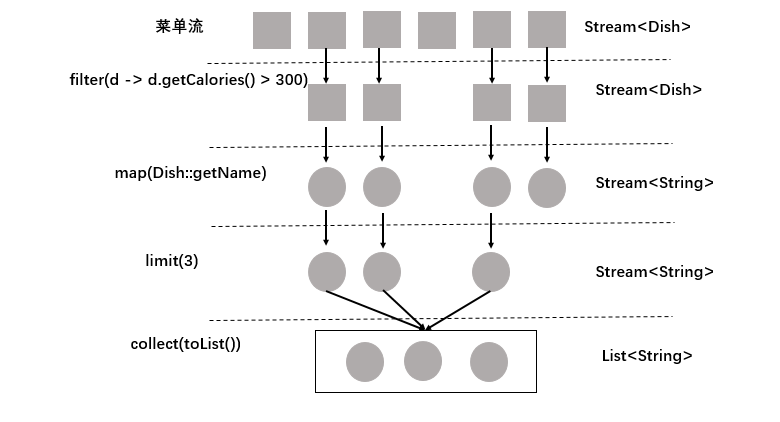
1、流水线：很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的很大的流水线。这让我们以后的一些优化成为了可能，如延迟和短路。流水线的操作可以可以看做对数据源进行数据库式查询。

2、内部迭代：与使用迭代器进行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：



在本例中，我们先是对menu调用stream方法，由菜单得到一个流。数据源是菜肴列表(菜单)，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，所有其他操作都会返回另一个流，从而构成一条流水线。最后，collect操作开始处理流水线，并且返回结果(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。下图显示了流操作的顺序：filter、map、limit、collect。



filter：接收Lambda表达式，从流中排除某些元素。在本例中，通过传递Lambda表达式d -> d.getCalories() > 300，选择出热量超过300卡路里的菜肴。

map：接收Lambda表达式，将元素转换成其他形式或者提取信息。在本例中，通过传递方法引用Dish::getName提取到了每道菜的菜名。

limit：截断流，使其元素个数不超过给定的数量。

collect：将流转换为其他形式。在本例中，流被转换成为一个列表。

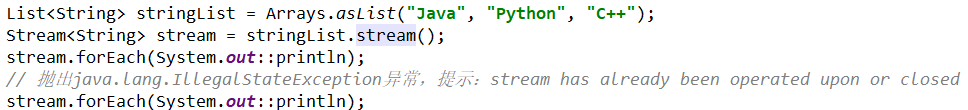
注意看，上面的这段代码与逐项处理菜单列表的代码有很大不同。首先，我们使用了声明的方式来处理菜单数据，即说明对这些数据要做什么，而并没有去实现筛选(filter)、提取(map)或者截断(limit)功能，而是交由Stream API来处理，由Stream API来决定如何优化这条流水线，例如筛选、提取和截断操作可以一次进行，并在找到三道菜后立即停止。

## 流与集合

粗略的说，集合与流之间的差异就在于什么时候进行计算。集合是内存中的数据结构，它是急切创建的，你可以往集合中添加或删除元素，但每个元素必须先计算出来才能添加到集合中或从集合中删除。流则是在概念上的数据结构，你不能往流中添加或删除元素，其元素是按照需要进行计算的。

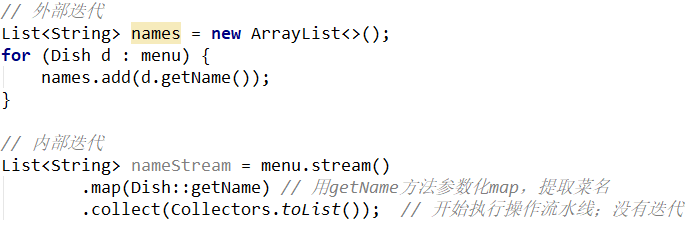
1、流只能遍历一次

注意，和迭代器类似，流只能遍历一次，遍历完成之后，这个流就被消费掉了。



2、流是内部迭代

使用Collection接口时，需要用户去迭代（比如for-each），这被称为外部迭代。而Stream使用内部迭代，即Stream帮你把迭代做了，并且把得到的流值存在了某个地方，你只需要说明要做什么就可以了。



内部迭代时，代码可以透明的并行处理，或者用更优化的顺序进行处理。如果用Java过去的外部迭代，则这些优化都是很困难的。Stream库的内部迭代可以自动选择一种适合你硬件的数据表示和并行实现。

## 流操作

java.util.stream.Stream接口定义了许多操作，它们可以分为两大类。看下之前的例子：

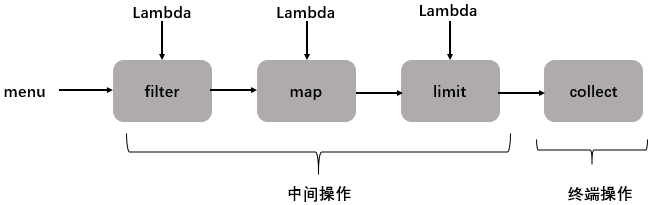
List<String> threeHighCaloricDishName = menu.stream()  
 .filter(d -> d.getCalories() > 300)  
 .map(Dish::getName)  
 .limit(3)  
 .collect(Collectors.*toList*());

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触发流水线执行并且关闭它。

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。下图展示了这两类的操作。



1、中间操作

诸如filter或者sorted等中间操作会返回一个流，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，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，我们把代码改了一下，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，这个编程风格要是搁在生成代码里面那就吓死人了，但是学习的时候却可以直接看清楚流的顺序)：

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(Collectors.toList());

System.out.println(names);

执行结果如下：

filtering: pork

mapping: pork

filtering: beef

mapping: beef

filtering: chicken

mapping: chicken

[pork, beef, chicken]

你会发现，有好几种优化利用了流的延迟性质。第一，尽管很多菜的热量都高于300卡路里，但只会选出了前三个，这是因为limit操作和一种被称为短路的技巧。第二，尽管filter和map是两个独立的操作，但它们合并到同一次遍历中了(我们把这种技术称为循环合并)。

2、终端操作

终端操作会生成结果，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是一个返回为void的终端操作，它会对源中的每道菜应用一个Lambda。

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

3、使用流

总而言之，流的使用一般包括三件事：

（1）一个数据源（如集合）来生成流

（2）一个中间操作链，形成一条流的流水线

（3）一个终端操作，执行流水线并生成结果

 流的流水线背后的理念类似于构造器模式。在构建器模式中有一个调用链用来设置一套配置(对流来说这就是一个中间操作链)，接着是调用built方法(对流来说就是终端操作)。

## 使用流

你已经看到了流让你从外部迭代转向内部迭代，这样，你就不用写下面这样的代码来显式的管理数据集合的迭代（外部迭代）了。

List<Dish> **vegetarianDishes** = **new** ArrayList<>();  
**for** (Dish d : menu) {  
 **if** (d.isVegetarion()) {  
 vegetarianDishes.add(d);  
 }  
}

你可以使用支持filter和collect操作的Stream API(内部迭代)管理对集合数据的迭代。你只需要将筛选条件为作为行为参数传递给filter方法就行了。

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

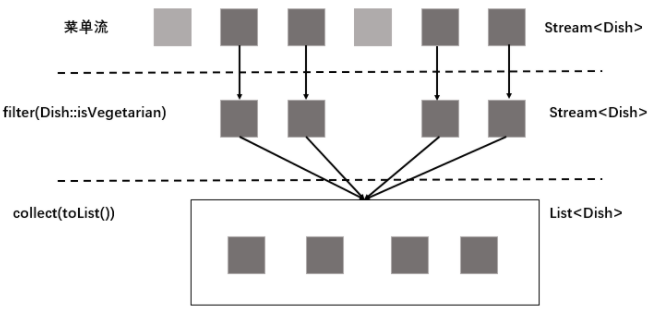
这种处理数据的方式很有用，因为你让Stream API管理如何处理数据，这样Stream API就可以在背后进行多种优化。此外，使用内部迭代的话，Stream API可以决定并行运行你的代码，这要是用外部迭代的话就办不到了，因为你只能用单一线程挨个迭代。

### 筛选和切片

#### 1、用断言筛选

Streams接口支持filter方法，该操作会接收一个断言作为参数，并返回一个包括所有正确断言的元素的流。例如，你可以像下图所示这样的，筛选出所有的素材，创建一张素食菜单：

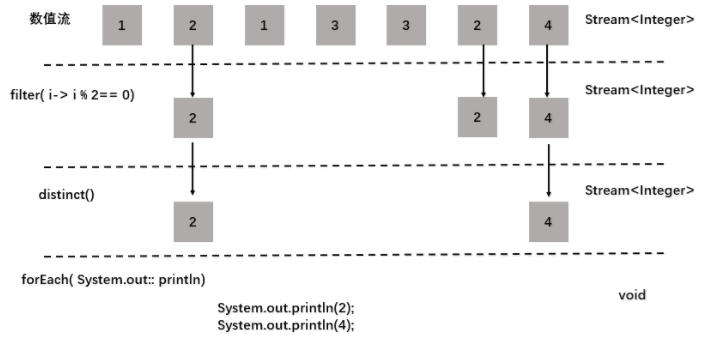
List<Dish> vegeratianMenu = menu.stream()  
 .filter(Dish::isVegetarion)  
 .collect(Collectors.*toList*());



#### 2、筛选各异的元素

 流还支持一个叫作distinct方法，它会返回一个元素各异(根据流所生成元素的hashCode和equals方法实现)的流。例如，一下代码会筛选出列表中所有的偶数，并且确保没有重复。下图直观地显示了这个过程。

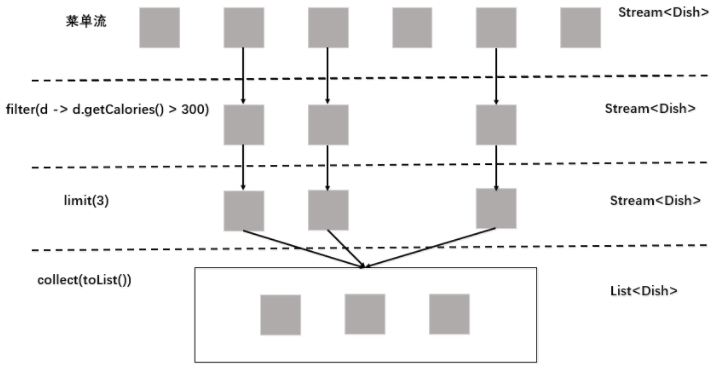
List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 1, 2, 3, 4, 5);  
numbers.stream()  
 .filter(i -> i % 2 == 0)  
 .distinct()  
 .forEach(System.***out***::println);



#### 3、截断流

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，选出热量超过300卡路里的头三道菜：

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

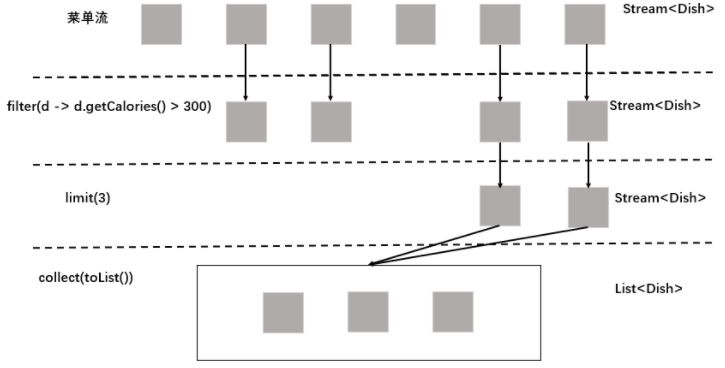


注意，limit也可以使用在无序流上，比如源是一个Set。这种情况下，limit的结果不会以任何顺序排列。

#### 4、跳过元素

流还支持skip(n)方法，返回一个扔掉前n个元素的流，如果流中元素个数不足n个，则返回一个空流。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。下图展示了这个查询。

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



### 映射

#### 1、对流中每一个元素应用函数

流还支持map方法，它会接收一个函数式接口作为参数，这个函数函数式接口会被应用到每个元素上，并将其映射成一个新的元素。例如，下面的代码把方法引用Dish::getName传给了map方法，来提取流中的菜肴的名称：

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

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

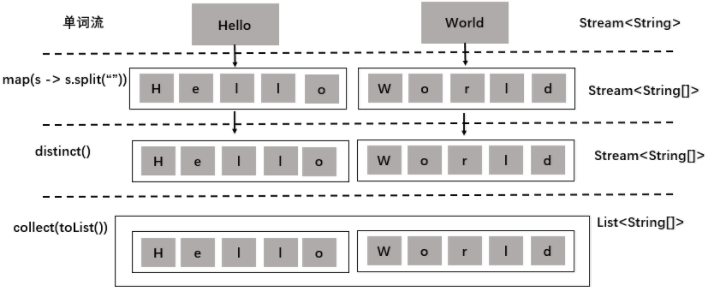
#### 2、流的扁平化

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。让我们拓展一下，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：

List<String> words = Arrays.*asList*(**"Hello"**, **"World"**);  
List<String[]> list = words.stream()  
 .map(w -> w.split(**""**))  
 .distinct()  
 .collect(Collectors.*toList*());  
  
**for** (**int** i = 0, n = list.size(); i < n; i++) {  
 String[] strings = list.get(i);  
 **for** (**int** j = 0, m = strings.**length**; j < m; j++) {  
 System.***out***.println(strings[j]);  
 }  
}

这个方法的问题在于，传递给map方法的Lambda为每个单词返回了一个String[](String类型的数组)。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<String>来表达式一个字符串，下图说明了这个问题：·



那怎么解决这个问题呢？Stream提供了flatMap来解决该问题。

首先，你需要一个字符流，而不是数组流。有一个叫Arrays.stream()的方法可以接收一个数组并产生一个流，例如：

String[] arrayOfWords = {**"Hello"**, **"World"**};  
Stream<String> streamOfWords = Arrays.*stream*(arrayOfWords);

将以上操作用在流水线中，看看会发生什么：

List<String> anotherWords = Arrays.asList("Hello", "World");

List<Stream<String>> anotherList = anotherWords.stream()

.map(word -> word.split("")) // Stream<String[]>

.map(Arrays::stream) // Stream<String>，但有多个

.distinct() // Stream<String>

.collect(Collectors.toList()); // List<Steam<String>>

当前的解决方案依然无法解决问题，因为你现在得到的是一个Stream<String>集合。

接下来使用flatMap解决这个问题。

List<String> yetAnotherWords = Arrays.asList(**"Hello"**, **"World"**);  
List<String> yetAnotherList = yetAnotherWords.stream()  
 .map(word -> word.split(**""**)) // Stream<String[]>  
 .flatMap(Arrays::stream) // Stream<String>，只有一个  
 .distinct() // Stream<String>  
 .collect(Collectors.toList()); // List<String>

使用flatMap的效果是：各个数组并不是分别映射成为一个流，而是将所有单个流合并起来，扁平化为一个流。

#### 3、查找和映射

Stream API提供了allMatch、anyMatch、noneMatch、findMatch和findAny方法来查看数据集中的元素是否匹配一个给定的属性。

（1）检查断言中是否至少匹配一个元素

anyMatch用于判断：流中是否有一个元素能够满足给定的断言。比如，你可以使用它来看看菜单里面里面是否含有素食可选择：

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

anyMatch返回一个boolean类型，因此是一个终端操作。

（2）检查断言是否匹配所有元素

allMatch用于判断：流中的元素是否全部满足给定的断言。比如，你可以使用它来看看菜单的热量是否小于1000卡路里。

**if** (menu.stream().allMatch(d -> d.getCalories() < 1000)) {  
 System.***out***.println(**"The menu is healthy."**);  
}

与allMatch相对的是noneMatch，它可以用于判断流中的元素是否全部不满足给定的断言。

 对于流而言，某些操作(例如：allMatch、anyMatch、noneMatch、findFirst和findAny)不用处理整个流就能得到结果。同样，limit也是一个短路操作：它只需要创建一个给定大小的流，而用不着处理流中所有的元素。在碰到无限大小的流的时候，这种操作就有用了：它们可以把无限流变成有限流。短路操作是Java中的&&和||运算符短路在流中的版本。

（3）查找元素

findAny用于返回当前流中的任意元素，它可以与其他流操作结合使用。比如你可能想找到任意一个素食菜肴，那么结合使用filter和findAny就可以实现这个效果。

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

findAny会返回一个Optional对象。java.util.Optional<T>类是一个容器类，代表一个值存在或者不存在。Java8的库设计人员引入了Optional<T>，这样就不用返回众所周知容易出问题的null了。Optional<T>提供的类方法如下：

boolean isPresent()：有值时返回true，否则返回false

void ifPresent(Consumer<T> block)：在值存在的时候执行给定的代码

T get()：会在值存在时返回值，否则会抛出一个NoSuchElement异常

orElse(T other)：会在值存在时返回值，否则返回一个默认值

（4）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或者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();

你可能会想，为什么会同时有findFirst和findAny呢？这是因为并行的缘故。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上限制较少。

### 归约

使用reduce操作可以表达更加复杂的查询，比如“计算菜单中的总卡路里”或者“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。这样的查询可以被归类为归约操作(将流归约成为一个值)，用函数式编程语言的术语来说，这就是折叠(fold)，因为你可以将这个操作看成把一张长长的纸(流)反复折叠成一个方块，而这就是折叠操作的结果。

#### 1、元素求和

在看reduce方法之前，先看看如何使用for-each循环对数字列表中的元素求和：

List<Integer> someNumbers = Arrays.*asList*(1, 2, 3, 4, 5);

**int** sum = 0;  
**for** (**int** i : someNumbers) {  
 sum += i;  
}

someNumbers中的每个元素都用加法运算符反复迭代来得到结果。通过反复使用加法，你把一个数字列表归约成一个数字，代码中有两个参数：

  A.总和变量的初始值，在这里是0；

  B.将列表所有元素结合在一起的操作，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，岂不是很好？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 = someNumbers.stream().reduce(0, (a, b) -> a + b);

reduce接收两个参数：

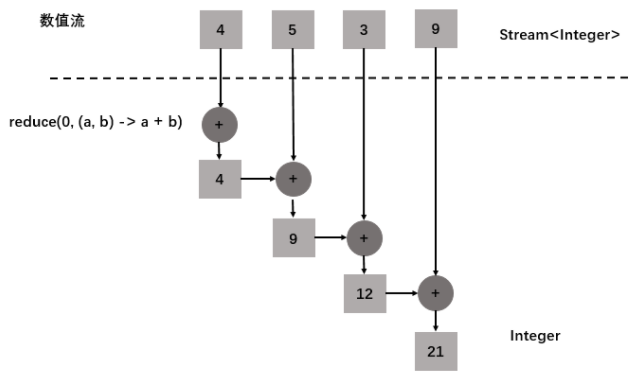
A.一个初始值，这里是0；

B.一个BinaryOperator<T>来将两个元素结合起来产生一个新值，这里我们用的是Lambda表达式 (a,b) -> a + a。

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 produt = someNumbers.stream().reduce(0, (a, b) -> a \* b);

下图展示了reduce操作是如何作用于一个流的：Lambda反复结合每个元素，直到流被归约成一个值。



让我们深入研究一下reduce操作是如何对一个数字流求和的。首先，0作为Lambda(a)的第一个参数，从流中获得4作为第二个参数(b)，0 + 4得到4，它成了新的累计值。然后再用累积值和流中下一个元素5调用Lambda，产生新的累计值9。接下来，再用累积值和下一个元素3调用Lambda，得到12。最后，用12和流中最后的一个元素9调用Lambda，得到最终结果21。

你可以使用方法引用让这段代码更加的简洁。在java8中，Integer类现有一个静态的sum方法来对两个参数求和，这恰好是我们想要的， 用不着反复用Lambda表达式写同一段代码：

int produt = someNumbers.stream().reduce(0, Integer::sum);

reduce还有一个重载方法，它不接收初始值，但是会返回一个Optional对象：

Optional<Integer> optional = someNumbers.stream().reduce(Integer::sum);

为什么它会返回一个Optional<Integer>呢？这是因为考虑流中没有任何元素的情况。reduce操作无法返回其和，因此将结果包括在一个Optional对象中，以表明可能不存在。

#### 2、最大值和最小值

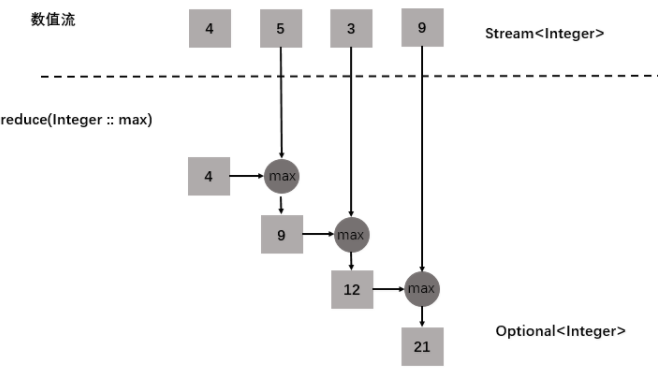
正如你前面看到的，reduce接收两个参数：

  A.一个初始值

  B. 一个Lambda来把两个流元素结合在一起来并且产生一个新值。

因此，要计算最大值，你需要一个给定两个元素能够返回最大值的Lambda，reduce操作会考虑新值和流中下一个元素，并产生一个新的最大值，直到整个流程消耗完了。你可以像下面这样使用reduce来计算流中的最大值，如图所示。

Optional<Integer> max = someNumbers.stream().reduce(Integer::*max*);



要计算最小值，你需要把Integer::min传给reduce来替换Integer::max。

Optional<Integer> min = someNumbers.stream().reduce(Integer::*min*);

#### 3、归约方法的优势

相比于前面的逐步迭代求和，使用reduce的好处在于：这里的迭代被内部迭代抽象掉了，这让内部实现得以选择并行执行reduce操作，而迭代式求和则要更新共享变量sum，这不是那么容易并行化的，如果你加入了同步，你很可能会发现线程竞争抵消了并行本应带来的性能提升，当然，你也可以使用fork/join来进行并行化：将输入分块，分块求和，最后再合并起来，但是这样的话代码看起来就完全不一样了。如果你使用流来对所有的元素并行求和，你的代码几乎不用修改，只用将stream()换成parallelStream()即可。

Integer min = someNumbers.parallelStream().reduce(0, Integer::min);

但是要并行执行这段代码也要付出一定代价：传递给reduce的Lambda表达式不能更改状态(如实例变量)，而且操作必须满足结合律才可以按任意顺序执行。

### 流状态

诸如map或者filter等操作会从输入流中获取每一个元素，并在输入流中得到结果。这些操作一般都是无状态的，即它们没有内部状态。

 但是诸如reduce、sum、max等操作需要内部状态来累积结果，例如求和时需要一个int值暂存求和值。

另外，诸如sort或distinct等操作一开始都和filter和map差不多，都是接收一个流，再生成一个流(中间操作)，但有一个关键的区别：从流中排序和删除重复项时都需要知道先前的历史，例如，排序要求所有的元素都放入缓冲区才能给输出流加入一个元素，这一操作的存储要求是无界。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 操作 | 类型 | 返回类型 | 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、有界 | Stream<R> | Comparator<R> | (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 |  |  |

### 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数据类型，再进行求和。要是可以直接像下面调用sum方法就好了：

**int** calories = menu.stream()  
 .map(Dish::getCalories)

.sum();

但是这是不可能的，因为map方法会生成一个Stream<T>，虽然流中的元素是Integer类型，但是Streams接口没有定义sum方法。但不要担心，Stream API还提供了原始类型流特化，专门支持处理数值流的方法。

#### 1、原始类型特化流

Java8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。每个接口都有常用的数值归约的新方法，比如对数值流求和的sum，找到最大元素的max，此外还有在必要时再把他们转换回对象的方法。要记住的是，这些特化的原因并不在于流的复杂性，而是装箱造成的复杂性，即类似int和Integer之间的效率差异。

（1）映射到数值流

将流转换为特换版本的常用是mapToInt、mapToDouble和mapToLong。这些方法和前面说的map方法的工作方式一样，只是他们返回的是一个特化流，而不是Stream<T>。例如，你可以像下面这样用mapToInt对menu中的卡路里总和：

**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)，并且返回一个IntStream(而不是Stream<Integer>)。然后你就可以调用IntStream接口定义的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。

（2）转换回对象流

同样，一旦有了数值流，你可能会想将其转换回非特化流。要把原始流转换为一般流(每个int都会装箱成一个Integer)，可以使用boxed方法，如下所示：

IntStream intStream = menu.stream().mapToInt(Dish::getCalories);  
Stream<Integer> boxed = intStream.boxed();

（3）默认值OptionalInt

前面我们介绍了Optional类，这是一个可以表示值存在或者不存在的容器。Optional可以用Integer、String等参数类型来参数化。对于三种原始流特化，也分别有一个Optional原始数据类型特化版本：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(0);

#### 2、数值范围

  和数字打交道时，有一个常用的东西就是数值范围，比如你想要生成1到100之间的所有数字。Java8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收起始值，第二个参数接收接收结束值。但是rang是不包含结束值的，而rangeClosed则包含结束值。让我们来看一个例子：

IntStream eventNumbers = IntStream.*rangeClosed*(1, 100)  
 .filter(n -> n % 2 == 0);  
System.***out***.println(eventNumbers.count());

## 构建流

到目前为止，你已经能够使用stream方法将集合变成流。此外，我们还介绍了如何根据数值范围创建数值流。但事实上，创建流的方式还有很多。

### 由值创建流

你可以使用静态方法Stream.of来显式的创建一个流，它可以接收任意数量的参数。例如，以下代码直接使用Stream.of创建了一个字符流：

Stream<String> stream = Stream.of("Java", "Python", "C++");

你可以使用empty方法得到一个空流，如下：

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。它接收一个数组作为参数。例如，你可以使用将一个原始类型int的数组转换成一个IntStream，如下所示：

**int**[] numbers = {1, 2, 3, 4, 5};  
**int** sum = Arrays.*stream*(numbers).sum();

### 由文件创建流

java.nio.file.Files中的很多静态方法都会返回一个流，例如File.lines会返回一个由指定文件中的各行构成的字符串流。你可以用这个方法看看这个文件有多少个不相同的词：

**long** uniqueWords = 0;  
String filePath = **"C:\\Users\\kevin\\Desktop\\测试.txt"**;  
**try** (Stream<String> lines = Files.*lines*(Paths.*get*(filePath))) {  
 uniqueWords = lines.map(line -> line.split(**" "**))  
 .flatMap(Arrays::*stream*)  
 .distinct()  
 .count();  
} **catch** (IOException e) {  
 e.printStackTrace();  
}

你可以使用Files.lines得到一个流，其中的每个元素都是给定文件中的一行。然后，你可以对line调用splie方法将行拆分成一个单词，需要注意的是，你应该利用flatMap产生一个扁平的单词流，而不是给每一行生成一个单词流对象。最后，把distinct和count方法链接起来，统计流有多少各不相同的单词。

### 由函数生成流

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，以避免打印无穷个值。

#### 1、迭代

我们先来看一个iterate的简单例子，然后解释：

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

iterate方法接收一个初始值(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<T>类型)。这里，我们使用Lambda表达式 n -> n+2,返回的是前一个元素加上2。因此，iterate方法生成了一个所有正偶数的流：流的第一个元素是初始值0。然后加上2来生成新的值2，再加上2来得到新的值4，以此类推。这种iterate操作基本上是顺序的，因为结果取决于前一次的结果。请注意，此操作将生成一个无限流。我们使用limit方法来显示限制流的大小，这里只选择了前10个偶数。然后可以调用forEach终端操作来消费流，并且分别打印每个元素。

   一般来说，在需要依次生成一系列的值的时候应该使用iterate方法，比如一系列日期：1月31日，2月1日，以此类推。

#### 2、生成

 与iterate方法类似，generate方法也可以让你按需生成一个无限流，但是generate不是依次对每个新生成的值应用一个函数，而是接收一个Supplier<T>类型的Lambda来提供新值。我们先来看一个简单的用法：

Stream.*generate*(() -> (**int**) (Math.*random*() \* 10))  
 .limit(10)  
 .sorted(Integer::compareTo)  
 .forEach(System.***out***::println);

  这段代码将生成流，流中的元素是0-10之间的随机数。Math.random静态方法被用作新值生成器，limit方法用于限制流的大小，否则流会无限长。

  你可能会想到，generate方法还有什么用途。我们使用的供应源(指向Math.random的Lambda表达式)是无状态的：它不会在任何地方记录新值以备以后计算使用。但是供应源不一定是无状态的，你可以创建存储状态的供应源，你可以修改状态，并在为流生成下一个值时使用。举个例子，我们将利用generate创建斐波那契数列，但需要注意的是，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。

首先你需要做的是创建一个IntSupplier，它要把前一项的值保存在状态中，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的状态。下面的代码就是如何创建一个调用时返回下一个斐波那契的IntSupplier：

IntSupplier fib = **new** IntSupplier() {  
 **private int previous** = 0;  
 **private int current** = 1;  
  
 **@Override  
 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);

上面的代码创建了一个IntSupplier的实例，此对象有两个实例变量，其中记录了前一个斐波那契和当前的斐波那契，getAsInt方法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。

## 收集器

### 引入

我们先来看一个不使用收集器的例子。假设你有一个Transaction的构成的List，想在你想按照货币种类进行分组，如果不使用收集器，你会写如下代码：

Map<Currency, List<Transaction>> transactionsByCurrencies = **new** HashMap<>();  
List<Transaction> transactions = **new** ArrayList<>();  
  
**for** (Transaction t : transactions) {  
 Currency currency = t.getCurrency();  
 List<Transaction> tmp = transactionsByCurrencies.get(currency);  
 **if** (tmp == **null**) {  
 tmp = **new** ArrayList<>();  
 transactionsByCurrencies.put(currency, tmp);  
 }  
 tmp.add(t);  
}

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，尽管将其翻译成白话就是：把列表中的交易按货币分组。

如果使用收集器，你就可以用一句话实现完全相同的结果：

Map<Currency, List<Transaction>> transactionsGroupByCurrency =  
 transactions.stream().collect(Collectors.*groupingBy*(Transaction::getCurrency));

### 简介

前一个例子清楚地展示了函数式编程相对于指令式编程的一个主要优势：你只需要指出希望的结果，即做什么，而不用操心执行的步骤，即怎么做。要是多级分组，指令式和函数式编程之间的区别就会更加明显：由于需要好多层嵌套循环和条件，指令式代码很快就变得更难阅读、更难维护、更难修改，相比之下，函数式版本只要再加上一个收集器就可以轻松地增强功能了，可以看到，函数式API的另一个好处是更易复用和重用。

收集器非常有用，对流调用collect方法将对流中的元素触发一个归约操作（由Collector来参数化）。一般来说，Collector会对元素应用一个转换函数（很多时候是不体现任何效果的恒等转换，例如toList），并将结果累积在一个数据结构中，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，将交易本身累积在生成的Map中。

Collectors工具类提供了很多静态工厂方法，可以方便地创建常见的收集器实例，这些收集器主要提供了三大功能：

1、将流元素归约和汇总为一个值

2、元素分组

3、元素分区

### 归约

#### 1、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。你可以创建一个Comparator来根据所含热量对菜肴进行比较，并把它传递给Collectors.maxBy：

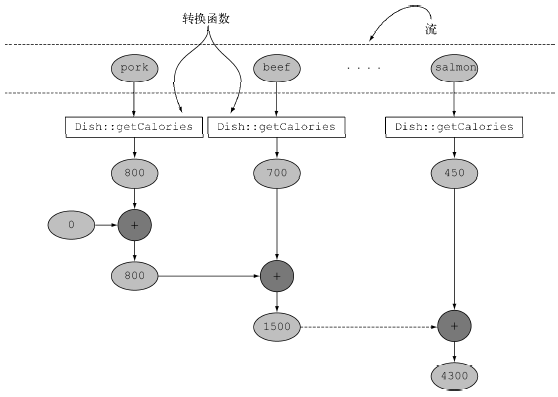
Comparator<Dish> dishCaloriesComparator =  
 Comparator.*comparingInt*(Dish::getCalories);  
Optional<Dish> mostCalorieDish = menu.stream()  
 .collect(Collectors.*maxBy*(dishCaloriesComparator));

#### 2、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ToIntFunction类型的函数，并返回一个收集器；该收集器在传递给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(Collectors.*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（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。Collectors.averagingInt、Collectors.averagingLong、Collectors.averaginDouble分别用于计算数值的平均数。

不过很多时候，你可能想要通过一次操作就得到最大值、最小值、平均值等所有结果，此时你就可以使用Collectors.summarizingInt方法。例如，通过一次summarizing操作你可以就数出菜单中元素的个数，并得到菜肴热量总和、平均值、最大值和最小值：

IntSummaryStatistics menuStatistics = menu.stream()  
 .collect(Collectors.*summarizingInt*(Dish::getCalories));  
System.***out***.println(menuStatistics);

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=4200, min=120, average=466.666667, max=800}

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

#### 3、连接字符串

joining工厂方法返回的收集器会将流中的每一个对象应用toString方法并将结果连接成一个字符串，这意味着你把菜单中所有菜肴的名称连接起来，如下所示：

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

joining在内部使用了StringBuilder来把生成的字符串逐个追加起来，如果Dish类有一个toString方法来返回菜肴的名称，那你就无需使用Dish::getName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());

另外，joining工厂方法还提供了两个重载版本：

joining(CharSequence delimiter)

joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

你可以使用它们来指定连接字符串的分隔符以及生成的字符串的前缀和后缀。

#### 4、广义的归约汇总

事实上，我们已经讨论的所有收集器，都是一个可以用reducing工厂方法定义的归约过程的特殊情况而已，Collectors.reducing工厂方法是所有这些特殊情况的一般化。可以说，先前讨论的案例仅仅是为了方便程序员而已。

Collectors.reducing工厂方法有几个重载方法：

reducing(BinaryOperator<T> op)

reducing(T identity, BinaryOperator<T> op)

reducing(U identity, Function<? **super** T, ? **extends** U> mapper, BinaryOperator<U> op)

对于第三个重载方法，其中第一个参数是归约操作的起始值，第二个参数是转换函数，第三个参数是一个BinaryOperator，将两个元素累积成一个同类型的值。对于第一个重载方法，它可以看做是第三个重载方法的特殊情况，即没有起始值，转换函数是恒等函数（即一个函数仅仅返回其输入参数），第三个参数是一个BinaryOperator。

你可以用三参数的reducing方法创建的收集器来计算你菜单的总热量，如下所示：

int totalCalories = menu.stream()  
 .collect(Collectors.*reducing*(0, Dish::getCalories, Integer::*sum*));

你也可以用单参数的reducing方法找到热量最高的菜，如下所示：

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

### 分组

#### 1、单级分组

一个常见的数据库操作是根据一个或多个属性对集合中的项目进行分组，就像前面讲到按货币对交易进行分组的例子一样。如果用指令式风格来实现的话，这个操作可能会很麻烦、啰嗦而且容易出错，但是，如果用Java 8所推崇的函数式风格来重写的话，就很容易转化为一个非常容易看懂的语句。比如，你要把菜单中的的菜按照类型进行分类，有肉的放一组，有鱼的放一组，其他的都放另一组，使用Collectors.groupingBy工厂方法返回的收集器可以很轻松地完成这项任务，如下所示：

Map<Dish.Type, List<Dish>> dishesByType = menu.stream()  
 .collect(Collectors.*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，这是个分类函数，用来把流中的元素分组。groupingBy分组操作的结果是一个Map，把分组函数返回的值作为映射的键，把流中所有具有这个键的元素列表作为对应的映射值。在上面的例子中，键就是菜的类型，值就是包含对应类型的菜肴列表。

分类函数不一定像方法引用那样可用，此时你就需要手写Lambda表达式了。例如，你可能想把热量不到400卡路里的菜划分为“低热量”（diet），热量400到700卡路里的菜划为“普通”（normal），高于700卡路里的划为“高热量”（fat）。

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

#### 2、多级分组

要实现多级分组，你可以使用由双参数版本的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。

Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =  
 menu.stream().collect(Collectors.*groupingBy*(Dish::getType,  
 Collectors.*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”，值是流中元素构成的List。

双参数版本的Collectors.groupingby的第二个收集器不一定要是另一个groupingBy，而可以是任何类型。例如，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(Collectors.*groupingBy*(Dish::getType,  
 Collectors.*counting*()));

其结果是下面的Map：

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

再举一个例子，你要按照菜肴的类型分类，查找各类菜肴中热量最高的菜肴：

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

这个分组的结果显然是一个map，以Dish的类型作为键，以包装了该类型中热量最高的Dish的Optional<Dish>作为值：

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

注意到上面这个例子中，Map的值是Optinal<Dish>，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，这个类型根本就不会出现在Map的键中，这意味着Optional包装器在这里不是很有用，所以你可能想要把它去掉。此时，你可以使用Collectors.collectingAndThen工厂方法返回的收集器，该工厂方法接收两个参数：要转换的收集器以及转换函数，并返回另一个收集器。

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

最后，常常和groupingBy联合使用的另一个收集器是Collectors.mapping方法生成的，该方法接收两个参数：一个函数对流中的元素做转换，另一个则将转换的结果收集起来。例如，你想要知道对于每种类型的Dish，菜单中都有哪些CaloricLevel。

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

我们在mapping方法中将生成的CaloricLevel流传递给一个toSet收集器，它和toList类似，不过是把流中的元素累积到一个Set而不是List中，以便仅保留各不相同的值。得到的Map结果如下：

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

上面的例子中，对于返回的Set是什么类型没有任何保证，你可以通过使用Collectors.toCollection方法并传递一个构造函数引用来执行返回的类型。

### 分区

分区是分组的特殊情况：由一个断言（返回一个布尔值的函数）作为分类函数，这个分类函数被称为分区函数。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，如果你是素食者或是请了一位素食的朋友来共进晚餐，可能会想要把菜单按照素食和非素食分开：

Map<Boolean, List<Dish>> partitionMenu = menu.stream()  
 .collect(Collectors.*partitioningBy*(Dish::isVegetarion));

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食菜肴了：

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

注意，用同样的分区断言对菜单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());

另外，partitioningBy也有一个重载方法：

partitioningBy(Predicate<? **super** T> predicate, Collector<? **super** T, A, D> downstream)

比如你想要对素食和非素食的菜肴按照类型进行分组，你可以像下面这样传递第二个收集器：

Map<Boolean, Map<Dish.Type, List<Dish>>> vegeratianDishesByType =  
 menu.stream().collect(  
 Collectors.*partitioningBy*(Dish::isVegetarion,  
 Collectors.*groupingBy*(Dish::getType))  
 );

结果如下：

{false={FISH=[prawns, salmon], MEAT=[pork, beef, chicken]},

true={OTHER=[french fries, rice, season fruit, pizza]}}

### Collectors类的静态工厂方法

import static java.util.stream.Collectos;

|  |  |  |
| --- | --- | --- |
| 工厂方法 | 返回类型 | 作用 |
| toList | List<T> | 把流中所有元素收集到一个List |
| 示例：List<Dish> dishes = menu.stream().collect(toList()); | | |
| toSet | Set<T> | 把流中所有元素收集到一个Set，删除重复项 |
| 示例：Set<Dish> dishes = menu.stream().collect(toSet()); | | |
| toCollection(Supplier<C>) | Collection<T> | 把流中所有元素收集到指定的集合 |
| 示例：Collection<Dish> dishes = menu.stream().collect(toCollection(), ArrayList::new); | | |
| counting() | Long | 计算流中元素的个数 |
| 示例：long howManyDishes = menu.stream().collect(countint()); | | |
| summingInt(ToIntFunction<? **super** T> mapper) | Integer | 对流中元素的一个整数属性求和 |
| 示例：int totalCalories = menu.stream().collect(summingInt(Dish::getCalories); | | |
| averagingInt(ToIntFunction<? **super** T> mapper) | Double | 对流中元素的一个整数属性求平均值 |
| 示例：double avgCalories = menu.stream().collect(averagingInt(Dish::getCalories)); | | |
| summarizingInt(ToIntFunction<? **super** T> mapper) | IntSummaryStatistics | 收集流中元素的整型属性的统计值，包括最大值、最小值、平均值和总和 |
| 示例：IntSummaryStatistics = menu.stream().collect(summarizingInt(Dish::getCalories); | | |
| joining() | String | 连接对流中每个元素调用toString()方法所生成的字符串 |
| 示例：String shortMenu = menu.stream().collect(joinging(", ")); | | |
| maxBy(Comparator<? **super** T> comparator) | 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，如果流为空，则返回Optional.empty() |
| 示例：Optional<Dish> fattest = menu.stream().collect(comparingInt(Dish::getCalories)); | | |
| minBy(Comprator<? super T> comparator) | 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，如果流为空，则返回Optional.empty() |
| 示例：Optioal<Dish> lightes = menu.stream().collect(comparingInt(Dish::getCalories)); | | |
| reducing(U identity, Function<? **super** T, ? **extends** U> mapper, BinaryOperator<U> op) | U | 使用转换函数将流中元素进行转换，然后使用一个初始值作为累加器的其实质，利用BinaryOperator与转换后的每个元素逐个结合，最终归约为单个值 |
| 示例：int totalCalories = menu.stream().collect(reducing(0, Dish::getCalories, Integer::sum)); | | |
| collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher) | RR | 包裹另一个收集器，对其结果应用转换函数 |
| 示例：int howManyDishes = menu.stream().collect(collectingAndThen(toList(), List::size)); | | |
| groupingBy (Function<? **super** T, ? **extends** K> classifier) | Map<K, List<T>> | 根据元素的一个属性值对流中的元素进行分组，并将属性值作为结果Map的键 |
| 示例：Map<Dish.Type, List<Dish>> dishesByType = menu.stream().collect(groupingBy(Dish::getType)); | | |
| partitioningBy (Predicate<? **super** T> predicate) | Map<Boolean, List<T>> | 根据对流中每个元素应用断言得到的结果来对元素进行分区 |
| 示例：Map<Boolean, List<Dish>> vegetarianDishes = menu.stream().collect(partitioningBy(Dish::isVegetarian)); | | |

### 收集器接口

接下来我们以toList工厂方法生成的收集器为例，仔细研究这个收集器是如何实现的，然后实现一个toListCollector<T>类。

Collector接口定义如下：

public interface Collector<T, A, R> {

Supplier<A> supplier();

BiConsumer<A, T> accumulator();

Function<A, R> finisher();

BinaryOperator<A> combiner();

Set<Characteristics> characteristics();

}

其中各个泛型参数的含义如下：

T是归约操作的输入元素的类型。

A是归约操作的累加器类型。

R是归约操作的结果类型。

下面我们来一个个分析Collector接口中声明的五个方法。

1、建立新的结果容器：supplier方法

supplier方法必须返回一个空的Supplier，在调用时会它会创建一个空的累加器实例，以供数据收集过程使用。在我们的ToListCollector中，supplier返回一个空List，如下所示：

public Supplier<List<T>> supplier() {

return () -> new ArraysList<T>();

}

你也可以使用构造函数引用：

public Supplier<List<T>> supplier() {

return ArrayList::new;

}

2、将元素添加到结果容器：accumulate方法

accumulate方法会返回执行归约操作的函数。当遍历到流中第n个元素时，这个函数在执行时会有两个参数：保存归约结果的累加器（已收集了流中的前n-1个项目），以及第n个元素本身。该函数将返回void。对于toListCollector，这个函数仅仅是把当前元素添加到已经遍历过的元素列表。

public BiConsumer<List<T>, T> accumulator() {

return (list, item) -> list.add(item);

}

你也可以使用方法引用：

public BiConsumer<List<T>, T> accumulator() {

return List::add;

}

3、对结果容器进行最终转换：finisher方法

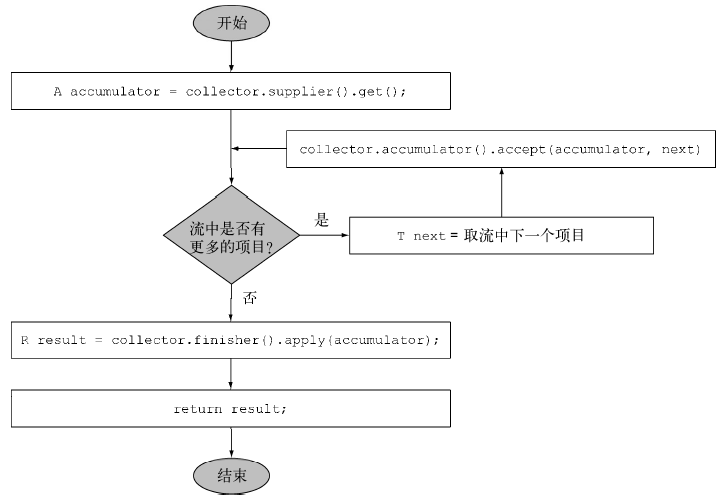
在遍历完流后，finisher方法必须返回一个函数，以便将累加器对象转换为整个集合操作的最终结果。对于toListCollector，由于累加器对象恰好符合预期的最终结果，因此无需对其进行转换，只需要返回identity函数：

public Function<List<T>, List<T>> finisher() {

return Function.identity();

}

这三个方法已经足以对流进行顺序归约，其执行流程图如下。但实践中的实现细节则更复杂一些，一方面是因为流的延迟性质，可能在collect操作之前还需要完成其他中间操作，另一方面则是理论上可能要进行并行归约。



4、合并两个结果容器：combiner方法

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toListCollector而言，只需要把从流的第二个部分收集到的元素列表加到遍历第一部分时得到的列表后面即可。

public BinaryOperator<T> combiner() {

return (list1, list2) -> {

list1.addAll(list2);

return list1;

}

}

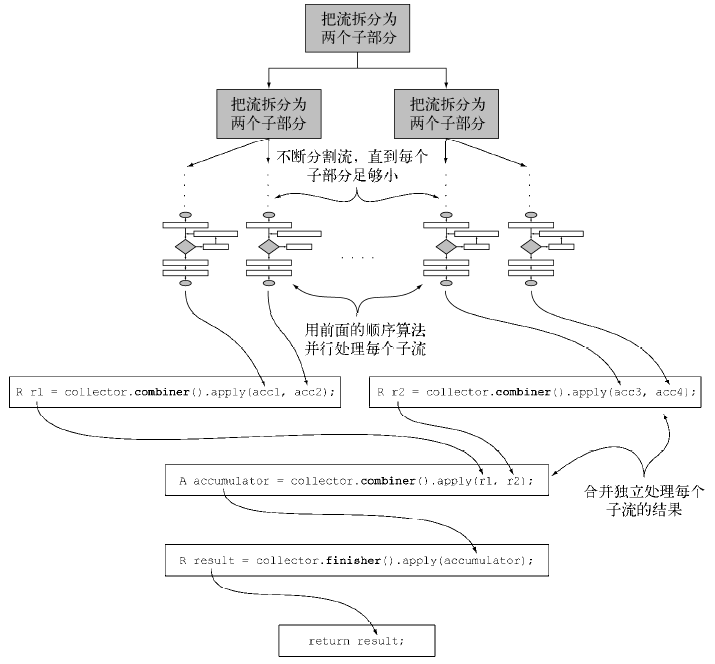
有了这四个方法，就可以对流进行并行归约了，过程如下：

（1）原始流会以递归方式拆分为子流，知道定义流是否需要进一步拆分的条件为false（如果分布式工作单位太小，并行计算往往比顺序计算慢，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）。

（2）现在所有的子流都可以进行并行处理了，对每个子流使用前图所示的顺序归约算法。

（3）最后，使用收集器combiner方法返回的函数，将所有的部分结果合并，这样会把原始流每次拆分时得到的子流对应的结果合并起来。

整个处理过程如下图所示。



5、characteristics方法

characteristics方法会返回一个不可变的Characteristics集合，它定义了用于优化归约操作的提示。Characteristics是一个包含三个元素的枚举。

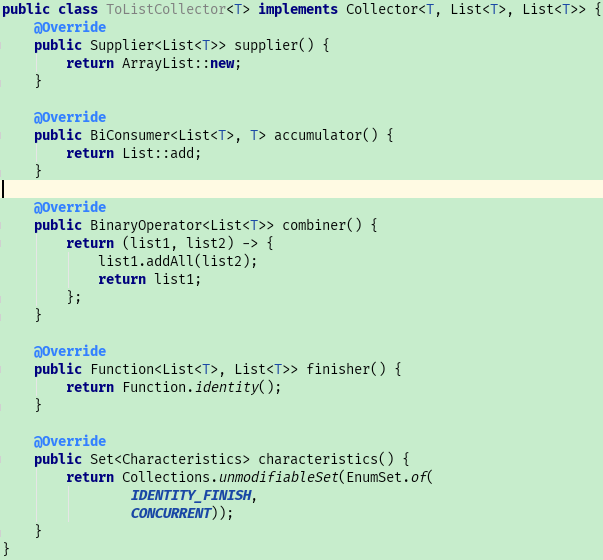
UNORDERED：归约结果不受流中元素的遍历和累积顺序的影响。

CONCURRENT：accumulator函数可以被多个线程同时调用。如果收集器不是UNORDERED，那它仅在用于无序数据源时才可以并行归约。

IDENTITY\_FINISH：finisher函数返回的是一个恒等函数，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不加检查的转换为结果R是安全的。

我们的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了，不用进一步转换。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，但我们刚才说过了，仅仅在背后的数据源无序时才会并行处理。

接下来我们可以开发自己的ToListCollector了，代码如下：



对于IDENTITY\_FINISH的收集操作，还有种方法可以得到相同的结果而无需从头实现新的Collector接口。Stream有一个重载的collect方法可以接收另外三个参数：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。因此，我们可以像下面这样把菜肴流中的元素收集到一个List中：

List<Dish> dishes2 = menu.stream().collect(  
 ArrayList::**new**, *// supplier* ArrayList::add, *// accumulator* List::addAll *// combiner*);

即便这样，我们认为，虽然这种形式比之前的写法更加紧凑和简洁，但不易读。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。另外值得注意的是，这个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。

### 开发收集器

假如你想要将前n个自然数划分为质数和非质数，代码可能会这样写：

**public** Map<Boolean, List<Integer>> partitionPrimes(**int** n) {  
 **return** IntStream.*rangeClosed*(2, n).boxed()  
 .collect(*partitioningBy*(candidate -> isPrime(candidate)));  
}  
  
**private boolean** isPrime(Integer candidate) {  
 **int** candidateRoot = (**int**) Math.*sqrt*(candidate);  
 **return** IntStream.*rangeClosed*(2, candidateRoot)  
 .noneMatch(i -> candidate % i == 0);  
}

有没有更好的优化方案呢？有，你可以仅仅处理被测试数是不是能够被质数整除的情况，如果除数本身都不是质数，就不用测试了。（定理：任意一个合数可以被分解为多个质数的乘积）。因此，我们可以仅仅用被测试数之前的质数来进行测试。为了做到这一点，我们需要开发一个收集器，因为目前所见的预定义收集器在收集过程中无法访问部分结果，这意味着，当测试某一个数字是否是质数时，你没法访问目前已经找到的其他质数的列表。

假设你有这个列表，那就可以把它传给isPrime方法，将方法重写如下：

**public static boolean** isPrime(List<Integer> primes, **int** candidate) {  
 **return** primes.stream().noneMatch(i -> candidate % i == 0);  
}

而且还应该应用先前的优化，即仅仅使用小于被测数平方根的质数来测试。你或许会想到用filter(p -> p <= candidateRoot)来筛选出小于被测数平方根的质数，但filter要处理整个流才能返回恰当的结果。因此，我们需要自己写一个方法，对于给定的排序列表和一个断言，返回元素满足断言的最长前缀：

**public static** <T> List<T> takeWhile(List<T> list, Predicate<T> p) {  
 **int** i = 0;  
 **for** (T t : list) {  
 **if** (!p.test(t)) {  
 **return** list.subList(0, i);  
 }  
 i++;  
 }  
 **return** list;  
}

此时的isPrime方法如下：

**public static boolean** isPrime(List<Integer> primes, **int** candidate) {  
 **int** candidateRoot = (**int**) Math.*sqrt*(candidate);  
 **return** *takeWhile*(primes, i -> i <= candidateRoot)  
 .stream()  
 .noneMatch(p -> candidate % p == 0);  
}

有了这个新的isPrime，你就可以实现自己的收集器了。

1、定义Collector类的签名

Collector接口定义是：

public Interface Collector<T, A, R>

其中T、A和R分别是流中元素的类型、用于累积部分结果的对象类型以及collect操作最终的结果类型。这里应该收集Integer流，而累加器和结果类型则都是Map<Boolean， List<Integer>>。

**public class** PrimeNumbersCollector **implements** Collector<Integer,  
 Map<Boolean, List<Integer>>,  
 Map<Boolean, List<Integer>>>

2、实现归约过程

接下来，你需要实现Collector接口中声明的五个方法。

supplier方法会返回一个在调用时创建累加器的函数：

**@Override  
public** Supplier<Map<Boolean, List<Integer>>> supplier() {  
 **return** () -> **new** HashMap<Boolean, List<Integer>>() {  
 {  
 put(**true**, **new** ArrayList<>());  
 put(**false**, **new** ArrayList<>());  
 }   
 };  
}

这里不仅创建了用作累加器的Map，还为true和false两个键下面初始化了对应的空列表，在收集过程中会把质数和非质数分别添加到这里。

accumulator方法定义了如何收集流中的元素，如下：

**@Override  
public** BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {  
 **return** (Map<Boolean, List<Integer>> acc, Integer candiate) -> {  
 acc.get(*isPrime*(acc.get(**true**), candiate))  
 .add(candiate);  
 };  
}

在这个方法中，你首先根据isPrime的结果获取质数或非质数列表，然后将被测数添加到相应的列表中。

3、让收集器并行工作（如果可能）

下一个方法要在并行收集时把两个部分的累加器合并起来，这里只需要合并两个Map，把第二个Map中的质数和非质数列表合并到第一个Map中的对应列表即可。

@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;  
 };  
}

需要注意的是，实际上这个收集器是不能并行使用的，因为该算法本身是顺序的，这意味着永远都不会调用combiner方法，你可以把它的实现留空，或者更好的做法是抛出一个UnsupportedOperationException异常。但为了让这个例子完整，我们还是决定实现它。

4、finisher方法和characteristic方法

accumulator正好就是收集器的结果，不用进一步转换，所以finisher方法就返回identity函数：

@Override  
**public** Function<Map<Boolean, List<Integer>>, Map<Boolean, List<Integer>>> finisher() {  
 **return** Function.*identity*();  
}

对于characteristic方法，它既不是CONCURRENT也不是UNORDERED，但却是IDENTITY\_FINISH的：

@Override  
**public** Set<Characteristics> characteristics() {  
 **return** Collections.*unmodifiableSet*(EnumSet.*of*(***IDENTITY\_FINISH***));  
}

现在你就可以使用自定义的收集器来完成一开始的功能了，如下；

**public** Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector(**int** n) {  
 **return** IntStream.*rangeClosed*(2, n).boxed()  
 .collect(**new** PrimeNumbersCollector());  
}

# 并行

在Java7之前，并行处理数据集非常麻烦。第一，你得明确地把包含数据的数据结构分为若干个部分。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。第四，等待所有线程完成，合并这些部分结果。

Java7引入了一个叫做Fork/Join的框架，让这些操作更稳定、更不易出错。

Java8中提供的Stream接口可以让你以声明性方法处理数据集，它可以将外部迭代换为内部迭代从而让原生Java库控制流元素的处理。这种方式可以让Java程序员无需显式实现优化来为数据集的处理加速，并且允许程序员对数据集执行操作流水线，从而自动利用计算机上的多个内核。

## 并行流

### 顺序流转换为并行流

你可以通过对数据源调用parallelStream方法来把集合转换成并行流。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，让它们都忙起来。让我们用一个简单的例子来试验一下这个思想。

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无穷大的数字流，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*);  
}

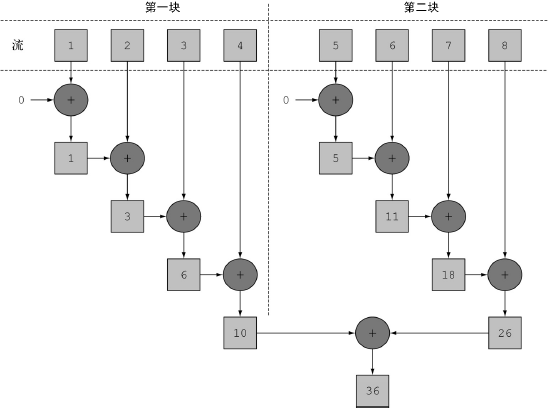
用更为传统的Java术语来说，这段代码与下面的迭代等价：

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

你可以调用parallel方法把顺序流转换为并行流，如下：

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

上述代码将数据源在Stream内部分成几块，然后对不同的块独立并行进行归纳操作，最后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。



需要注意的是，对顺序流调用调用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置了一个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。请注意，你可能以为把这两个方法结合起来，就可以更细化地控制在遍历流时哪些操作要并行执行，哪些要顺序执行。例如，你可以这样做：

stream.parallel()

.filter(…)

.sequential()

.map(…)

.parallel()

.reduce();

但最后一次parallel或sequential调用会影响整个流水线，在上述例子中，流水线会并行执行，因为最后调用的是它。

### 测试流性能

你可以开发一个方法来测试前面所写的顺序流和并行流的性能。

**public static long** measureSumPerf(Function<Long, Long> func, **long** n) {  
 **long** fastest = Long.***MAX\_VALUE***;  
 **for** (**int** i = 0; i < 10; i++) {  
 **long** start = System.*nanoTime*();  
 **long** sum = func.apply(n);  
 **long** duration = (System.*nanoTime*() - start) / 1\_000\_000;  
 System.***out***.println(**"Result: "** + sum);  
 **if** (duration < fastest) {  
 fastest = duration;  
 }  
 }  
 **return** fastest;  
}

这个方法接收一个函数func和一个long型变量n作为参数，它会对func执行10次并返回最短一次的执行时间。

假设你把先前开发的所有方法都放进了一个名为ParallelStreams的类，你就可以用如下代码来测试对前一千万个自然数求和要用多久：

System.***out***.println(**"Sequential sum done in: "** +

*measureSumPerf*(ParallelStreamTest::*sequentialSum*, 10\_000\_000) + **"msecs"**);  
System.***out***.println(**"Iterative sum done in: "** +

*measureSumPerf*(ParallelStreamTest::*iterativeSum*, 10\_000\_000) + **"msecs"**);  
System.***out***.println(**"Parallel sum done in: "** +

*measureSumPerf*(ParallelStreamTest::*parallelSum*, 10\_000\_000) + **"msecs"**);

结果如下：

Sequential sum done in: 91msecs

Iterative sum done in: 3msecs

Parallel sum done in: 97msecs

结果令人相当失望，求和方法的并行版本比顺序版本要慢很多，这是因为：

1、iterate生成的是装箱的对象，必须拆箱成数字才能求和。

2、你很难把iterate拆分成多个独立块来并行执行，因为每次应用这个函数都要依赖前一次应用的结果。这意味着整张数字列表在归纳过程开始时还没准备好，因为无法有效地把流划分为小块来并行处理。把流标记为并行，你其实是给顺序处理增加了开销，它还要把每次求和操作分到不同的线程上。

这说明并行编程可能很复杂，有时候甚至有点违反直觉，如果使用不当，它甚至可能让程序的整体性能更差。

那到底要怎么利用多核处理器，用流来高效地并行求和呢？你可以使用LongStream.rangeClosed方法来代替iterate方法，理由是：

1、LongStream.rangeClosed产生原始类型的long数字，没有装箱拆箱的开销。

2、LongStream.rangeClosed会生成数字范围，很容易拆分为独立的小块。

让我们先看一下它用于顺序流的性能如何，看看拆箱的开销到底要不要紧：

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

这一次的输出是：

Random sum done in: 4msecs

这个数值流比前面用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。

我们再对这个新版本应用并行流：

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

这一次的输出是：

Parallel ranged sum done in: 1msecs

可以看到，我们得到了一个比顺序执行更快的并行归纳。这表明，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，因此在使用并行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，比如使用的算法改变了某些共享状态。因此，要记住避免共享可变状态，确保并行Stream得到正确的结果。

### 高效使用并行流

下面给出一些定性意见，帮你决定某个特定情况下是否有必要使用并行流。

1、如果有疑问，测试。把顺序流转成并行流轻而易举，但却不一定是好事，因此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。

2、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，但凡有可能都应该用这些流。

3、有些操作本身在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。

4、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粗略的定性估计。*Q*值较高就意味着使用并行流时性能好的可能性比较大。

5、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。

6、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。

7、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。

8、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付出的代价就可能会超出通过并行流得到的性能提升。

下表按照分解性总结了一些数据源适不适合于并行。

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

最后，并行流背后使用的基础架构是Java7中引入的Fork/Join框架。

## Fork/Join框架

### 使用Fork/Join框架

Fork/Join框架的目的是以递归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。首先来看看如何定义任务和子任务。

Fork/Join框架的任务是提交到ForkJoinPool线程池中的，要把任务提交到这个池，必须创建，必须创建RecursiveTask<R>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。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拆分成子任务的逻辑，以及无法再拆分或不方便再拆分时，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 {

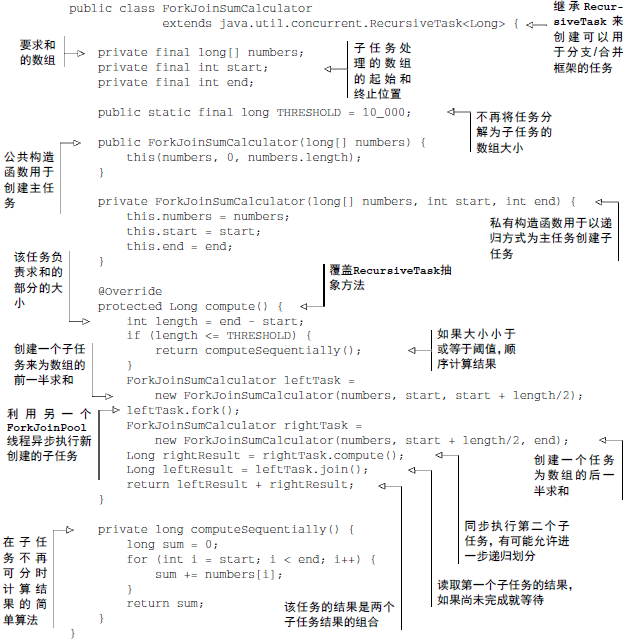
将任务分成两个子任务

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

}

这里举一个用Fork/Join框架的实际例子，还以前面的例子为基础。如前所述，你需要先为RecursiveTask类做一个实现：



测试方法如下：

**public static long** forkJoinSum(**long** n) {  
 **long**[] numbers = LongStream.*rangeClosed*(1, n).toArray();  
 ForkJoinTask<Long> task = **new** ForkJoinSumCalculator(numbers);  
 **return new** ForkJoinPool().invoke(task);  
}

测试结果如下：

ForkJoin sum done in: 19msecs

这个性能看起来比用并行流的版本要差，这只是因为必须先把数字流都放进一个long[]数组，之后才能在ForkJoinSumCalculator任务中使用它。

### 使用Fork/Join框架的最佳实践

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。

1、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，因为每个子任务都必须等待另一个子任务完成才能启动。

2、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。

3、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。

4、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Fork/Join计算上就不行了。

5、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。我们已经说过，一个任务可以分解成多个独立的子任务，才能让性能在并行化时有所提升。另外，在比较同一算法的顺序和并行版本的性能时还有别的因素要考虑。像任何其他Java代码一样，Fork/Join框架需要“预热”或者说要执行几遍才会被JIT编译器优化。这就是为什么在测量性能之前跑几遍程序很重要，我们的测试框架就是这么做的。同时还要知道，编译器内置的优化可能会为顺序版本带来一些优势（例如执行死码分析——删去从未被使用的计算）。

### 工作窃取

Fork/Join框架使用了工作窃取（work stealing）算法。每个线程都将分配给它的任务保存一个双向链式队列中，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。但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。这个过程一直继续下去，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。

## Spliterator

### 概述

Spliterator是Java 8中加入的另一个新接口，表示“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接口定义如下：

public interface Spliterator<T> {

boolean tryAdvance(Consumer<? super T> action);

Spliterator<T> trySplit();

long estimateSize();

int characteristics();

}

tryAdvance方法的行为类似于普通的Iterator，它会按顺序一个一个使用Spliterator中的元素，如果还有元素需要比那里则返回true。

trySplit方法是专门为Spliterator接口设计的，它可以把一些元素划分出去给第二个Spliterator，从而进行并行处理。

estimateSize方法用于估计还剩下多少元素需要遍历，因为即使不那么确切，但能快速算出来是一个值也有助于让拆分均匀一点。

characteristics方法将返回一个int，代表Spliterator的特性。

将Stream拆分成多个部分的算法是通过trySplit方法实现的，这是一个递归过程。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。这样不断对Spliterator调用trySplit直到它返回null，表明它处理的数据结构不能再分割。

拆分过程受到Spliterator本身的特性影响，下表中总结了这些特性。

|  |  |
| --- | --- |
| 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 |

### 自定义Spliterator

接下来我们实现一个Spliterator。假设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 static 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;  
}

测试用例如下：

**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"**);

结果如下：

Found 19 words

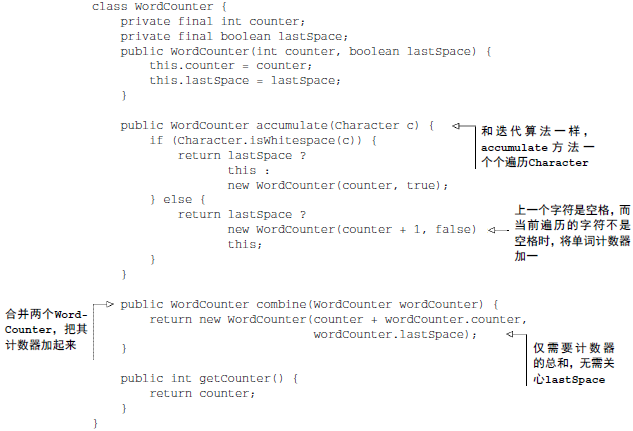
理想情况下，你会想要用更为函数式的风格来实现它，因为这样你就可以用并行Stream来并行化这个过程，而无需显式地处理线程和同步问题。

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

Stream<Character> stream = IntStream.range(0, SENTENCE.length())

.mapToObj(SENTENCE::charAt);

你可以对这个流做归约来计算字数。在归约流时，你得保留由两个变量组成的状态：一个int用来计算到目前为止数过的字数，还有一个boolean用来记得上一个遇到的Character是不是空格。因为Java没有元组（tuple，用来表示由异类元素组成的有序列表的结构，不需要包装对象），所以你必须创建一个新类WordCounter来把这个状态封装起来，如下所示。



接下来写一个方法来归约Character流即可：

**public static int** countWords(Stream<Character> stream) {  
 WordCounter wordCounter = stream.reduce(  
 **new** WordCounter(0, **true**),  
 WordCounter::accumulate,  
 WordCounter::combine);  
 **return** wordCounter.getCounter();  
}

测试用例如下：

Stream<Character> stream = IntStream.*range*(0, SENTENCE.length())  
 .mapToObj(SENTENCE::charAt);  
System.***out***.println(**"Found "** + *countWords*(stream) + **" words"**);

结果如下：

Found 19 words

你可以尝试用并行流来加快字数统计：

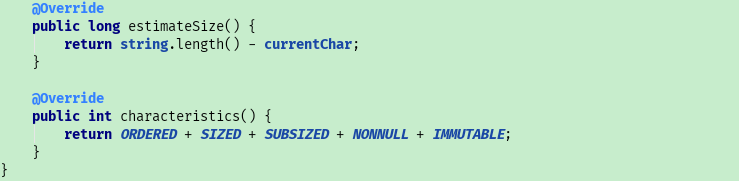
Stream<Character> parallelStream = IntStream.*range*(0, SENTENCE.length())  
 .mapToObj(SENTENCE::charAt).parallel();  
System.***out***.println(**"Found "** + *countWords*(parallelStream) + **" words"**);

结果如下：

Found 26 words

这是因为原始的String在任意位置拆分，所以有时一个词会被分为两个词，然后数了两次。这就说明，拆分流会影响结果，而把顺序流换成并行流就可能使结果出错。因此，你需要自定义一个Spliterator，确保String不是在随机位置拆开的，而只能在词尾拆开。





tryAdvance方法把String中当前位置的Character传给了Consumer，并让位置加1，如果新的指针位置小于String 的总长，说明还有要遍历的Character，则tryAdvance返回true。

trySplit方法中，如果剩余的Character数量低于下限，你就返回null表示无需进一步拆分。否则，要执行拆分，则把试探的拆分位置设在要解析的String块的中间，为了避免把词在中间断开，于是就往前找，直到找到一个空格。一旦找到了适当的拆分位置，就可以创建一个新的Spliterator来遍历从当前位置到拆分位置的子串。

estimatedSize方法中，还需要遍历的元素就是这个Spliterator解析的String的总长度与当前遍历的位置的差。

characteristic方法告诉框架这个Spliterator是ORDERED（顺序就是String中各个Character的次序）、SIZED（estimatedSize方法的返回值是精确的）、SUBSIZED（trySplit方法创建的其他Spliterator也有确切大小）、NONNULL（String中不能有为null 的Character ） 和IMMUTABLE （ 在解析String 时不能再添加Character，因为String本身是一个不可变类）的。

现在就可以用这个新的WordCounterSpliterator来处理并行流了，如下所示：

Spliterator<Character> spliterator = **new** WordCounterSpliterator(SENTENCE);  
Stream<Character> parallelStream2 = StreamSupport.*stream*(spliterator, **true**);  
System.***out***.println(**"Found "** + *countWords*(parallelStream2) + **" words"**);

结果如下：

Found 19 words