# 简介

## 为什么需要再次修改Java

多核CPU的兴起成为了不容回避的事实。涉及锁的编程算法不但容易出错，而且耗费时间。人们开发了java.util.concurrent包和很多第三方类库，试图将并发抽象化，帮助程序员写出在多核CPU上运行良好的程序。很可惜，到目前为止，我们的成果还远远不够。

开发类库的程序员使用Java时，发现抽象级别还不够。处理大型数据集合就是个很好的例子，面对大型数据集合，Java还欠缺高效的并行操作。开发者能够使用Java 8编写复杂的集合处理算法，只需要简单修改一个方法，就能让代码在多核CPU上高效运行。为了编写这类处理批量数据的并行类库，需要在语言层面上修改现有的Java：增加Lambda表达式。

面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因为这两种编程方式我们都得学。这种新的抽象方式还有其他好处。不是所有人都在编写性能优先的代码，对于这些人来说，函数式编程带来的好处尤为明显，程序员能编写出更容易阅读的代码——这种代码更多地表达了业务逻辑的意图，而不是它的实现机制。易读的代码也易于维护、更可靠、更不容易出错。

在写回调函数和事件处理程序时工程序员不必再纠缠于匿名内部类的冗繁和可读性，函数式编程让事件处理系统变得更加简单。能将函数方便地传递也让编写惰性代码变得容易，惰性代码在真正需要时才初始化变量的值。

Java 8还让集合类可以拥有一些额外的方法default方法。程序员在维护自己的类库时，可以使用这些方法。

## 什么是函数式编程

每个人对函数式编程的理解不尽相同。但其核心是：在思考问题时，使用不可变值和函数，函数对一个值进行处理，映射成另一个值。不理解什么是函数式编程这根本不 重要，我们关心的是如何写出好代码，而不是符合函数式编程风格的代码。

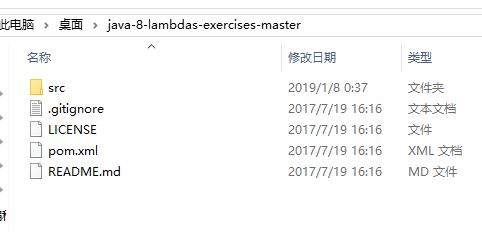
## 1.3 示例

本书中的示例全部都围绕一个常见的问题领域构造：音乐。具体来说，这些示例代表了在专辑上常常看到的信息，有关术语定义如下。

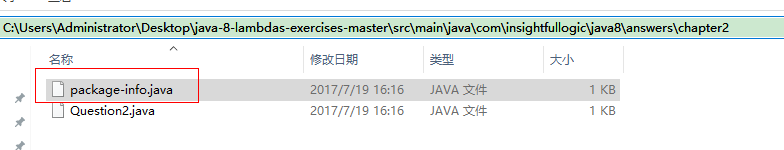
* Artist：创作音乐的个人或团队
  + name：艺术家的名字 (例如“甲壳虫乐队”)
  + members乐队成员 (例如“约翰·列侬”)，该字段可为空
  + origin乐队来自哪里(例如“利物浦”)
* Track：专辑中的一支曲目
  + name：曲目名称
* Album：专辑，由若干曲目组成
  + name：专辑名
  + tracks：专辑上所有曲目的列表
  + musicians：参与创作本专辑的艺术家列表

本书将使用这个问题讲解如何在正常的业务领域或者Java应用中使用函数式编程技术。

## 随书代码介绍



书中课后练习的问题答案看这个文件：



有些问对错或者简单代码的，会直接在这个文件给出答案，如果代码多的会单独写一个文件，它会告诉你文件名叫什么，如“Question2.java”，如果是一张图片，它也会说明，图片保存在：\java-8-lambdas-exercises-master\src\main\resources\com\insightfullogic\java8\examples

书中的示例代码保存在：\java-8-lambdas-exercises-master\src\main\java\com\insightfullogic\java8\examples目录下

# Lambda表达式

Java8的最大变化是引入了Lambda表达式：这是一种紧凑的、传递行为的方式。

## 2.1 第一个Lambda表达式

button.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

System.out.println("button clicked");

}

});

设计匿名内部类的目的是为了方便Java程序员将代码作为数据传递，不过，匿名内部类不够简洁，为了调用一行重要的逻辑代码，不得不加上4行冗繁的样板代码。而且这些代码相当难读懂，因为它没有清楚地表达程序员的意图。我们不想传入对象，只想传入行为，在Java8中，上述代码可以写成一个Lambda表达式，如下：

button.setOnClickListener(v -> System.out.println("button clicked"));

和传入一个实现接口的对象不同，我们传入了一段代码块——一个没有名字的函数。“v”是参数名，“->”将参数和Lambda表达式的主体分开，而主体是用户点击按钮时会运行的一些代码。

和使用匿名内部类的另一处不同在于声明v参数的方式。使用匿名内部类时需要显式地声明参数类型View v，而在Lambda表达式中无需指定类型，程序依然可以编译。这是因为javac根据程序的上下文(setOnClickListener方法的签名)在后台推断出了参数v的类型。这意味着如果参数类型不言而明，则无需显式指定。

注：尽管与之前相比，Lambda表达式中的参数需要的样板代码很少，但是Java 8仍然是一种静态类型语言。为了增加可读性并迁就我们的习惯，声明参数时也可以包括类型信息，而且有时编译器不一定能根据上下文推断出参数的类型！

Lambda表达式还可以这样用：

OnClickListener ocl = v -> System.***out***.println(**"button clicked"**);  
button.setOnClickListener(ocl);

所以可以这样理解：Lambda表达式是用于创建一个接口对象的一种简便方式。

button.setOnClickListener(v -> System.***out***.println(**"button clicked"**));

把一个Lambda表达式传给了setOnClickListener()方法，该方法接收的是一个OnClickListener对象，而我们传进去的Lambda表达式看起来不像是一个对象，而像是一个匿名方法，我们把一个方法像数据一样进行了传递。

所以，也可以这样理解，Lambda表达式就是一个匿名方法,将行为像数据一样进行传递。

## 2.2 如何辨别Lambda表达式

Lambda表达式除了基本的形式之外，还有几种变体，如下：

1、Runnable noArguments = () -> System.out.prlintln("Hello World");

2、OnClickListener oneArgument = v -> System.out.println("button clicked");

3、Runnable multiStatement = () -> {

System.out.print("Hello");

System.out.print(" World");

};

4、BinaryOperator<Long> add = (x, y) -> x + y;

5、BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;

一、1中所示的Lambda表达式不包含参数，使用空括号()表示没有参数。该Lambda表达式实现了Runnable接口，该接口只有一个run方法，没有参数，返回类型为void。

二、2中所示的Lambda表达式包含一个参数，可以省略参数的括号

三、3中所示的Lambda表达式的主体是一段代码块，使用{ }括起来，只有一行的代码可以省略花括号

四、4中所示的Lambda表达式包含2个参数，BinaryOperator的函数可以是有返回值的，则x + y的结果是返回值（如果只有一行代码，则不需要写return，否则必须写上return）。

五、5中所示的Lambda表达式显式地声明了参数的类型

Java中初始化数组时，数组的类型就是根据上下文推断出来的，如下：

String[] array = { “hello”, “world” }; // 等号右边并没有声明类型，系统根据上下文推断出类型信息

## 2.3 引用值，而不是变量

在内部类中引用外部的一个局部变量，这个局部变量必须是final的，意味着这个变量不能为其重复赋值，那就不应该算变量了，因为不会变了，就只是一个值而已了，所以内部类引用的是外部的一个值。

String name = **"button clicked"**;  
button.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 System.***out***.println(name);  
 }  
});

这段代码在Java8中不会报错，虽然name没有加上final，但是它已经相当于是final，编译器在编译的时候会自动加上final，这样的final变量称之为“既成实事上的final变量”，所以在Lambda表达式中使用的局部变量也必须是final类型的，如下将会报错：

String name = **"button clicked"**;  
name = **"haha"**;  
button.setOnClickListener(v -> System.***out***.println(name));

这里name进行了两次赋值，编译器没办法把它变成final，这种行为也解释了为什么Lambda表达式被称为闭包。未赋值的变量与周边环境隔离起来，进而被绑定到一个特定的值。

下面则是正确的：

String name = **"button clicked"**;// 其实这里加上final修饰也是可以的，但是既然不用加也可以就不写了，省事。写不写看个人爱好吧！  
button.setOnClickListener(v -> System.***out***.println(name));

## 2.4 函数接口

函数接口是指只拥有一个抽象方法的接口，用作Lambda表达式的类型。如：Runnable noArguments = () -> System.out.prlintln("Hello World");这里的Lambda表达式为“() -> System.out.prlintln("Hello World");”，它的类型为：Runnable。

Java开发工具包（JDK）中提供的一组核心函数接口会频繁出现 ，如下罗列了一些最重要的函数接口：‘

|  |  |  |
| --- | --- | --- |
| 接口 | 参数 | 返回类型 |
| Predicate<T> | T | boolean |
| Consumer<T> | T | void |
| Function<T,R> | T | R |
| Supplier<T> | None | T |
| UnaryOperator<T> | T | T |
| BinaryOperator<T> | (T, T) | T |

## 2.5 类型推断

什么时候需要显式地指明类型？应该出于让代码最便于阅读的目的去考虑，有时省略类型可以减少干扰更容易弄清状况；而有时却需要类型信息帮助理解代码。

Lambda表达式中的类型推断实际上是Java7就引入的目标类型推断的扩展。Map<String, String> map = new Map<String, String>(); 这行代码到了Java 7就可以使用类型推断：Map<String, String> map = new Map<>();编译器根据变量的声明就可以推断出new的Map的泛型类型，这没什么难度。

如果将构造函数直接传给一个方法，也可以根据方法签名来推断类型，如下：

public void useHashMap(Map<String, String> map);

useHashMap(new HashMap<>());

Java7中程序员可以省略构造函数的泛型类型，Java8更进一步，程序员可以省略Lambda表达式中的所有参数类型。需要注意的是，Java8中对类型推断系统的改善值得一提，上面的例子将new HashMap<>()传给useHashMap方法，即使编译器拥有足够的信息，也无法在java7中通过编译。

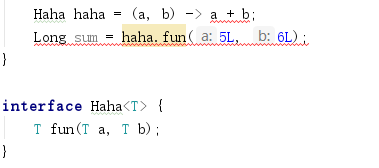
**interface** Haha<T> {  
 **void** fun(T a, T b);  
}

Haha<Long> haha = (a, b) -> System.***out***.println(**"a = "** + a + **", b = "** + b);  
haha.fun(5L, 6L);

如上例子，输出结果：a = 5, b = 6，通过类型推断可知函数a和b的类型是Long，如果我们在声明的时候不写泛型，如下：

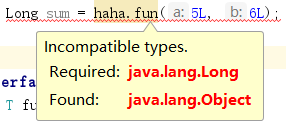
Haha haha = (a, b) -> System.***out***.println(**"a = "** + a + **", b = "** + b);

此时编译运行也是一样的结果，那这时a和b的类型是什么呢？答：Object。再修改例子，如下:



可以看到有两行代码报错了，错误描述分别如下：





因为在声明Haha的时候没有写明泛型，所以参数a和b是被系统推断为Object类型的，函数fun的返回类型也是Object类型。

第一条报错语句原因是两个Object对象是不能使用 + 这种操作符进行相加的。

第二条报错语句原因是fun返回的是Object，不能把一个Object赋值给Long类型的变量。

假设有一个接口A，一个接口B，这两个接口中各有一个方法，方法名不同，但是返回值和参数列表相同，示例如下：

i static interface A {

boolean aa(int param);

}

static interface B {

boolean bb(int param);

}

static class C {

void func(A a) { System.out.println(a.aa(1)); }

void func(B b) { System.out.println(b.bb(3)); }

}

public static void main(String[] args) throws Exception {

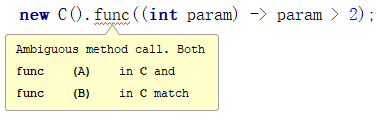
new C().func(param -> param > 2);

}

new C().func(i -> i != 100); 这行代码将报错，有两个，如下：



这个是属于工具问题，Lambada表达式是没问题的，我们给参数加上括号和类型即可解决，然后就剩下一个错误，如下：



大概意思为 ：模棱两可的方法调用，C里的 func(A)和C里的func(B)都匹配。

因为无法推断Lambda表达式是A类型还是B类型。所以在这种路种情况下应该显示的声明Lambda表达式类型，如下：

B b = param -> param > 2;

new C().func(b);

## 练习

ThreadLocal，Java8为该类新加了一个工厂方法，接受一个Lambda表达式并返回一个新的ThreadLocal对象，使用如下：

public class Looper {

public final static ThreadLocal<Looper> sThreadLocal = ThreadLocal.withInitial(() -> new Looper());

}

# 流

## 3.1 从外部迭代到内部迭代

如下代码计算从伦敦来的艺术家有多少个：

int count = 0;

for (Artist art : allArtists) {

if (art.isFrom("London")) count++;

}

尽管这样的操作可行，但存在几个问题，每次迭代集合时，都需要写很多样板代码，将for循环改造成并行方式也很麻烦，需要修改每个for循环才能实现。

此外，上述代码无法流畅传达程序员的意图。for循环的样板代码模糊了代码的本意。程序员必须阅读整个循环体才能理解。若是单一的for循环，倒也问题不大，但面对一个满是循环(尤其是嵌套循环)的庞大代码库时，负担就重了。

就其背后的原理来看，for循环其实是一个封装了迭代的语法糖，我们在这里多花点时间，看看它的工作原理。首先调用iterator方法，产生一个新的Iterator对象，进而控制整个迭代过程，这就是外部迭代。迭代过程通过显式调用Iterator对象的hasNext和next方法完成迭代。展开后的代码如例3-2所示，图3-1展示了迭代过程中的方法调。

例3-2 使用迭代器计算来自伦敦的艺术家人数

int count = 0;

Iterator<Artist> it = allArtists.iterator();

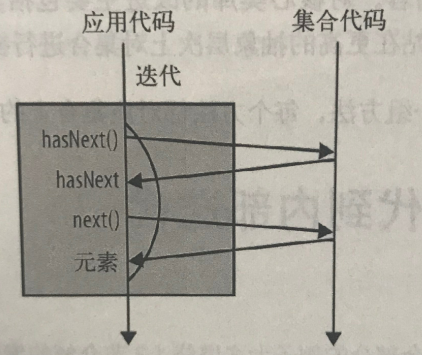
while (it.hasNext()) {

Artist art = iterator.next();

if (art.isFrom("London")) count++;

}

图3-1：外部迭代



然而，外部迭代也有问题。首先，它很难抽象出本章稍后提及的不同操作; 此外，它从本质上来讲是一种串行化操作。总体来看，使用for循环会将行为和方法混为一谈。

另一种方法就是内部迭代，如例3-3所示。首先要注意stream()方法的调用，它和例3-2中调用iterator()的作用一样，但是该方法不是返回一个控制迭代的Iterator对象，而是返

回内部迭代中的相应接口：Stream。

例3-3 使用内部迭代计算来自伦敦的艺术家人数

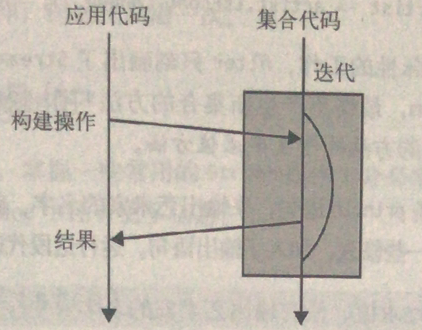
long count = allArtists.stream()

.filter(art -> art.isFrom("London"))

.count();

图3-2 展示了使用类库后的方法调用流程，与图3-1形成对比。

图3-2：内部迭代



通过两张图片的对比，一个是在“应用代码”的地方进行的迭代，一个是在“集合代码”的内部进行的迭代。

Stream是用函数式编程方式在集合类上进行复杂操作的工作。

例3-3可被分解为两步更简单的操作

·找出所有来自伦敦的艺术家，

·计算他们的人数。

每种操作都对应stream接口的一个方法。为了找出来自伦敦的艺术家，需要对stream对象进行过滤filter。过滤在这里是指“只保留通过某项测试的对象”。测试由一个函数完成，根据艺术家是否来自伦敦，该函数返回true或者false。由于Stream API的函数式编程风格，我们并没有改变集合的内容，而是描述出stream里的内容。count()方法计算给定Stream里包含多少个对象。

## 3.2实现机制

例3-3中，整个过程被分解为两种更简单的操作：过滤和计数，看似有化简为繁之嫌——

例3-1中只含一个for循环，两种操作是否意味着需要两次循环?事实上，类库设计精妙，只需对艺术家列表迭代一次。

通常，在Java中调用一个方法，计算机会随即执行操作比如，System.out.println("Hello world")会在终端上输出一条信息。stream里的一些方法却略有不同，它们虽是普通的Java方法，但返回的stream对象却不是一个新集合，而是创建新集合的配方。现在尝试思考一下例3-4中代码的作用，一时毫无头绪也没关系，稍后会详细解释。

例 3-4 只过滤，不计数

allArtists.stream()

.filter(art -> art.isFrom("London"));

这行代码并未做什么实际性的工作，filter只刻画出了Stream，但没有产生新的集合。像filter这样只描述Strean，最终不产生新集合的方法叫作惰性求值方法，而像count这样最终会从Strean产生值的方法叫作及早求值方法。

如果在过滤器中加入一条println语句，来输出艺术家的名字，就能轻而易举地看出其中的不同，如下：

allArtists.stream()

.filter(art -> {

System.out.println(art.getName());

return art.isFrom("London")

});

如上代码，运行后将看不到任何的输出，如果再调用一个拥有终止操作的流，如计数操作count()，就会看到艺术家的名字会被输出。可以这么理解：filter方法只是往Stream对象上保存了一个过虑的条件（Lambda表达式），但是并没有使用，当调用count函数时和进行集合的迭代，而且在迭代时使用之前保存的过虑条件。

判断一个操作是惰性求值还是及早求值很简单，只需看它的返回值。如果返回值是Stream，那么是惰性求值，如果返回值是另一个值或为空，那么就是及早求值。使用这些操作的理想方式就是形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果，这正是它的合理之处。

整个过程和建造者模式有共通之处。建造者模式使用一系列操作设置属性和配置，最后调用一个build方法，这时对象才被真正创建。

读者一定会问：“为什么要区分惰性求值和及早求值?”只有在对需要什么样的结果和操作有了更多了解之后，才能更有效率地进行计算。例如，如果要找出大于10的第一个数字，那么并不需要和所有元素去做比较，只要找出第一个匹配的元素就够了（不然所有元素都遍历一下就有点浪费了），这也意味着可以在集合类上级联多种操作，但迭代只需一次（也就是说在一次迭代中完成你需要做的多件事）。

## 3.3 常用的流操作

### 3.3.1 collect(toList())

collect(Collectors.toList())方法由Stream里的值生成一个列表，是一个及早求值操作。

Stream的of方法使用一组初始值生成新的Stream对象，其内部原理是使用Arrays.stream(T[] array)方法将数组变成Stream对象的：

List<String> list1 = Stream.*of*(**"a"**, **"b"**, **"c"**).collect(Collectors.*toList*());  
List<String> list2 = Arrays.*asList*(**"a"**, **"b"**, **"c"**);  
System.***out***.println(list1.equals(list2)); *// 结果为true*

### 3.3.2 map

如果有一个函数可以将一种类型的值转换成另外一种类型，map操作就可以使用该函数，将一个流中的值转换成一个新的流。

例3-8 使用for循环将字符串转换为大写

List<String> uppercaseList = **new** ArrayList<>();  
List<String> lowercaseList = Arrays.*asList*(**"a"**, **"b"**, **"hello"**);  
**for** (String s: lowercaseList) {  
 String uppercaseString = s.toUpperCase();  
 uppercaseList.add(uppercaseString);  
}

例3-9 使用map操作将字符串转换为大写

List<String> uppercaseList = Stream.*of*(**"a"**, **"b"**, **"hello"**)  
 .map(s -> s.toUpperCase())  
 .collect(Collectors.*toList*());

map()方法是返回了一个新的Stream对象，这个Stream对象中包含了大写的"A"、"B"、"HELLO"。再来理解开头的那段话，String的toUpperCase()函数可以将一种小写字母转换成大写字母，则map操作就可以使用toUpperCase()函数将一个流中的值转换成一个新的流。

注：map函数接收一个Function参数，这个接口里面有一个apply方法，声明为：R apply(T t); ，Lambda表达式的格式还是看得懂的，不明白的是系统是怎么推断出T和R都是String的。模拟的简化代码如下：

**public interface** MyFunction<T, R> {  
 R apply(T t);  
}

**public static class** MyStream<T> {  
 **public static** <T> MyStream<T> of(T... args) {  
 **return new** MyStream<>();  
 }  
  
 <R> MyStream<R> map(MyFunction<? **super** T, ? **extends** R> mapper) {  
 **return new** MyStream<R>();  
 }  
}

MyStream.*of*(**"haha"**)  
 .map(s -> s.toUpperCase());

MyStream.of("haha")，这里方法传了String，所以静态方法中的泛型T被实例化为String，MyStream类上的泛型也被实例化为String，如下：

**public static class** MyStream<T> {  
 **public static** <String> MyStream<String> of(String... args) {  
 **return new** MyStream<>();  
 }  
  
 <R> MyStream<R> map(MyFunction<? **super** T, ? **extends** R> mapper) {  
 **return new** MyStream<R>();  
 }  
}

of方法中的泛型T并不是类上面的泛型T，通过上面代码可知，of方法的返回值类型为：MyStream<String>，所以return new MyStream<>()中虽然没有显示地声明类的泛型，但是系统肯定知道是要创建MyStream<String>类型的对象，所以这里MyStream类上的泛型T也被实例化了，如下：

**public static class** MyStream<String> {  
 <R> MyStream<R> map(MyFunction<? **super** String, ? **extends** R> mapper) {  
 **return new** MyStream<R>();  
 }  
}

Fuction接口如下：

**public interface** MyFunction<? **super** String, ? **extends** R> {  
 ? **extends** R apply(? **super** String t);  
}

MyStream.of("haha")已经看完，接下来看.map(s -> s.toUpperCase()); 怎么理解这句代码呢？这是一句Lambda表达式，这个表达式代表了MyFunction中的apply方法，所以s代表了参数。不明白的是编译是怎么把参数s推断成String类型的？

MyStream.of("String").map(s -> s.toUpperCase()); 使用Lambda表达式，这个代码是OK的，使用我们以前的方式，如下：

MyStream.*of*(**"String"**).map(**new** Function<String, Object>() {  
 @Override  
 **public** Object apply(String s) {  
 **return** s.toUpperCase();  
 }  
});

上面的方法实现是通过开发工具自动生成的，根据泛型Function中的第一个泛型可以是String或者String的父类，所以可以把String替换成CharSequence，如下：

MyStream.*of*(**"String"**).map(**new** MyFunction<CharSequence, Object>() {  
 @Override  
 **public** Object apply(CharSequence s) {  
 **return** s.toUpperCase(); // 但是CharSequence并没有这个toUpperCase()方法，所以可以肯定的是编译器把这个泛型实例化成了String  
 }  
});

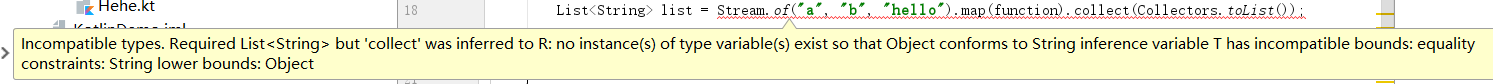
但是如果让开发工具自己去生成一个Function，他怎么能知道你第一个泛型是用String还是用String的哪个父类呢？可能就是因为不知道选哪个，所以编译器通过“? super String”这个泛型自动选择了String这个类型来当成第一个泛型的类型。我们把Function参数用到真实的代码，如下：

Function<String, Object> function = **new** Function<String, Object>() {  
 @Override  
 **public** Object apply(String s) {  
 **return** s.toUpperCase();  
 }  
};  
  
Stream.*of*(**"a"**, **"b"**, **"hello"**).map(function);

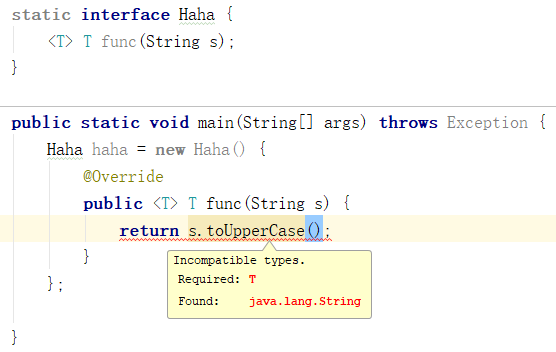
把最后一行代加上转变为集合，如下：

List<Object> list = Stream.*of*(**"a"**, **"b"**, **"hello"**).map(function).collect(Collectors.*toList*());

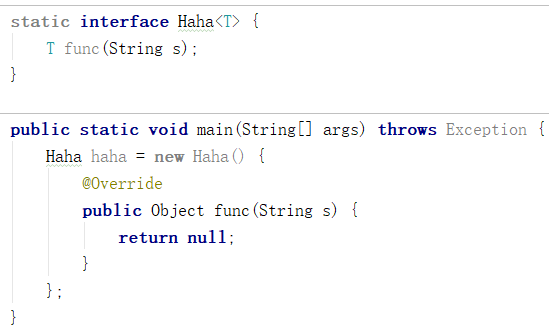
编译通过，修改List中的泛型为String，如下：



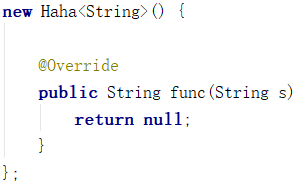
报错了，说明我们传的Function函数不符合要求，把Function函数中的第二个泛型参数也改成String，这时编译就OK了，运行也正常了。这说明对于map(s -> s.toUpperCase())，编译器给我们生成的Function对象中的两个泛型都是String，对于第一个泛型“? super String”可以是String或String的父类，编译器直接选择了String，那第二个泛型参数编译器是怎么推断成String的呢？因为apply方法有返回值，而s.toUpperCase()就是方法的返回值，所以编译器把s.toUpperCase()的类型String当成了返回类型。我们做实验如下：



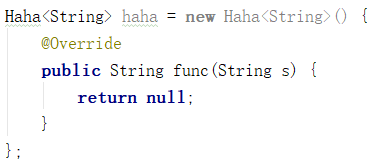
报错信息告诉我们“不兼容的类型，需要T，找到的是java.lang.String”，其实我们看Function类它的泛型是定义在类上面的，所以修改如下：



这就没报错了，这是因为泛型声明在类上面，我们在new Haha的时候没有指定泛型，则系统默认为Object，所以开发工具自动把返回类似创建为Object类型。因为类上有泛型，一般使用这个类的话要实例化类上的泛型，如下：



或者我们在声明Haha的时候指定泛型，则系统也会推断出返回类型是什么，如下：



接着试验：

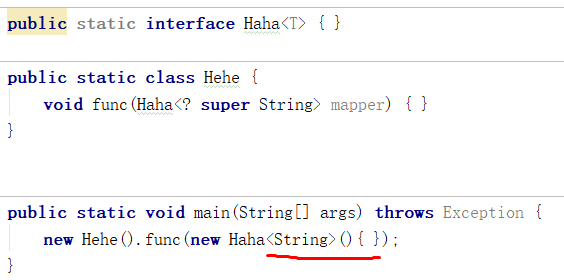
Haha<String> haha = s -> **"haha"**.toUpperCase();

这行代码编译通过，我们在声明Haha的时候使用了String类型，这充分说明了系统通过Lambda表达式给我们创建Haha对象时就是把泛型实例化为String了，不然这行代码没法编译通过的。我们知道Lambda表达式其实就是一种创建接口对象的方便方式，如果接口上有泛型，则在new的时候肯定就需要实例化泛型，所以在Lambda“s -> **"haha"**.toUpperCase();”这句表达式中，编译器通过我们的返回语句haha.toUpperCase()的类型，在创建 Haha时就把泛型实例化为这个类型。

#### 类型推断总结：

1. 在Lambda表达式中，编译器会把”? super String”这样类型的泛型直接实例化为String
2. 在Lambda表达式中，如果返回类型是泛型，则编译器会把泛型实例化为返回值一样的类型

关于第1点也可以实验，如下：



如上图，在new Haha的时候，后面红线的代码是由编译工具自动生成的，说明编译工具对于泛型“? Super String”直接取String做为泛型类型。

### 3.3.3 filter

找出集合中以数字打头的元素：

List<String> list = Arrays.*asList*(**"aaa"**, **"acb"**, **"1ff"**, **"e"**, **"2ccc"**);  
List<String> numberStartList = **new** ArrayList<>();  
**for** (String s : list) {  
 **if** (s.charAt(0) >= **'0'** && s.charAt(0) <= **'9'**) {  
 numberStartList.add(s);  
 }  
}  
numberStartList.forEach(e -> System.***out***.println(e));

使用Stream中的filter解决：

List<String> numberStartList = Stream.*of*(**"aaa"**, **"acb"**, **"1ff"**, **"e"**, **"2ccc"**)  
 .filter(s -> s.charAt(0) >= **'0'** && s.charAt(0) <= **'9'**)  
 .collect(Collectors.*toList*());

### 3.3.4 flatMap

List<Integer> list1 = Arrays.*asList*(1, 2, 3);  
List<Integer> list2 = Arrays.*asList*(4, 5, 6);  
List<List<Integer>> list3 = Stream.*of*(list1, list2).collect(Collectors.*toList*());  
list3.forEach(e -> System.***out***.println(e));

打印如果如下：

[1, 2, 3]

[4, 5, 6]

如果想把两个List合并，可使用flatMap方法，如下：

List<Integer> list1 = Arrays.*asList*(1, 2, 3);  
List<Integer> list2 = Arrays.*asList*(4, 5, 6);  
List<Integer> list3 = Stream.*of*(list1, list2).flatMap(list -> list.stream()).collect(Collectors.*toList*());  
System.***out***.println(list3);

打印结果如下：

[1, 2, 3, 4, 5, 6]

Stream.of(list1, list2) ：Stream中有两个列表

flatMap(list -> list.stream()) ：把Stream中的两个列表转换为两个Stream，然后将这两个Stream链接成一个Stream

### 3.3.5 max和min

Stream上的常用操作之一是求最大值和最小值。

求名字最短的人：

List<Person> list = Arrays.*asList*(**new** Person(**"黄蓉"**), **new** Person(**"西门吹雪"**), **new** Person(**"欧阳峰"**));

Person p = list.stream()

.min(Comparator.*comparing*(person -> person.**name**.length()))

.get();

System.***out***.println(p.**name**);

Java 8 给Comparator提供了一个新的静态方法comparing，用于方便地实现一个比较器，只需要告诉编译器拿什么作为比较即可。这个例子中是拿名字的长度作为比较。

Stream的min方法返回的是optional对象，Optional对象有点陌生，它代表一个可能存在也可能不存在的值。如果stream为空，那么该值不存在，如果不为空，则该值存在。先不必细究，4.10节将详细讲述optional对象，现在唯一需要记住的是，通过调用get方法可以取出optional对象中的值。max方法的使用与min一样。

花点时间研究一下comparing方法是值得的。实际上这个方法接受一个函数并返回另一个函数。我知道，这听起来像句废话，但是却很有用。这个方法本该早已加入Java标准库，但由于匿名内部类可读性差且书写冗长，一直未能实现。现在有了Lambda表达式，代码变得简洁易懂。

min方法的源码如下：

**public interface** Stream<T> {

**public final** Optional<T> min(Comparator<? **super T**> comparator) {  
 **return** reduce(BinaryOperator.*minBy*(comparator));  
}

}

Stream在创建的时候已经把T实例化为Person，所以：

**public interface** Stream<Person> {

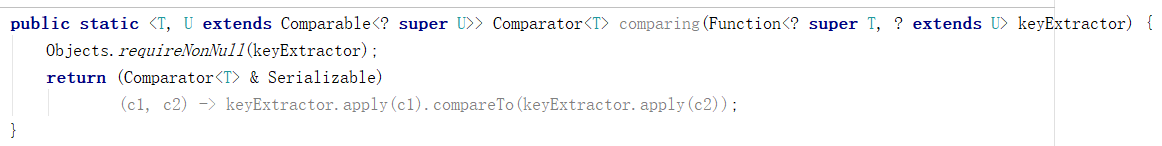
**public final** Optional<Person> min(Comparator<? **super** Person> comparator) {  
 **return** reduce(BinaryOperator.*minBy*(comparator));  
}

}

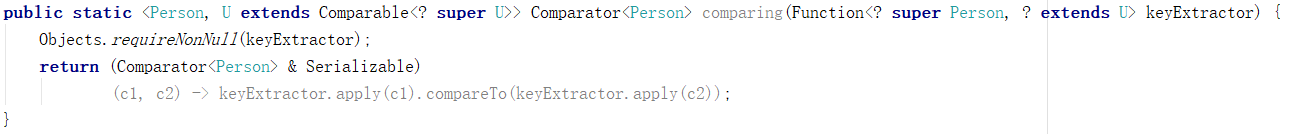
所以min方法的传参相当于：

Comparator<? super Person> comparator = Comparator.comparing(person -> person.name.length());

再来看comparing方法的源码如下：



这个方法返回的Comarator是带泛型的，而我们左边声明的Comparator表明了接收的活泛型为：？super Person，根据以前的经验，编译器会推断为Person，所以可以这样看源吗：



不太理解这里的Function的第一个泛型为什么要限定为可以是Person的父类。

再看comparing方法中参数keyExtractor的传叁：

Function<? super Person, U extends Comparable<? super U>> keyExtractor = person -> person.name.length();

可以这样理解：Function的第一个泛型被实例化为Person，第二个泛型被实例化为Comparable，这样再理解Function源码，如下：

**public interface** Function<Person, Comparable> {  
 Comparable apply(Person t);  
}

根据我们的Lambda表达式（person -> person.name.length()）,编译器为我们创建的Function参数为 ：

Function<? **super** Person, ? **extends** Comparable> keyExtractor = **new** Function<Person, Comparable>() {  
 @Override  
 **public** Comparable apply(Person person) {  
 **return** person.**name**.length();  
 }  
};

所以Function的apply方法返回的对象是Integer，而Integer是实现了 Comparable接口的。



这是一个Lambda表达式，用来创建Comparator<Person>对象的，这个Lambda表达式就是实现了Comparator中的compare方法，所以c1，c2会被推断为Person类型，方法返回一个int值，而Integer的compareTo方法也是返回int。所以可以简单理解成如下：

Comparator<Person> comparator = **new** Comparator<Person>() {  
 @Override  
 **public int** compare(Person o1, Person o2) {  
 **return i**nteger1.*compareTo*(integer2);  
 }  
};

再来看comaring方法的返回值过程：

Comparator<Person> comparator = (Comparator<Person> & Serializable) (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));

我们先理解(Comparator<> & Serializable)，这是类型强转，表示强转为Comparator 和Serializable，且必须同时能强转为这两种类型，否则会抛类转换异常，实验如下：

static class A<T> {}

static class B<T> extends A<T> { }

public static void main(String[] args) throws Exception {

B<String> b = new B<>();

Object c = b;

A<String> a = (A<String> & Serializable) b;

Serializable s = (A<String> & Serializable) b;

}

运行时将会抛出类转换异常，在第一次强转时就会抛出。让B实现Serializable即可解决问题。

不明白的是，通过Lambda表达式（(c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2))）由编译创建的Comparator对象是怎么会实现Seriailizable的 ？，难道是因为有Serializable的强转操作符，所以编译器在创建Comparator对象时就自动实现Serializable接口了？试验如下：

**public static interface** Haha<T> {  
 **void** func();  
}  
  
**public static void** main(String[] args) **throws** Exception {  
 Serializable s = (Haha<String> & Serializable) () -> System.***out***.println(**"hello!"**);  
 Haha<String> h = (Haha<String>) s;  
 h.func();  
}

运行通过，实验证明编译器看到有Serializable的强转操作，在创建Haha对象的时候自动实现了Serializable接口。

接下来看comparing方法中的泛型，先看Function的源码，如下：

**public interface** Function<T, R> {  
 R apply(T t);  
}

在comparing函数中声明了 Function变量，在声明Function变量时限定Function的第一个泛型可以是T或者T的父类，限定了第二个泛型可以是U或者U的子类，而这个U本身被限定为必须是Comparable或者继承自Comparable，而Comparable中的泛型为可以是Comparable或者Comparable的父类，这一点很奇葩，Comparable的父类只有Object，这样理解是错的，因为U可能是Comparable的子类，则这个子类它的父类应当是Comparable，所以可以是自己或者父类。

### 3.3.6 通用模式

max和 min方法都属于更通用的一种编程模式。要看到这种编程模式，最简单的方式是使用for循环重写前面查找最短人名的例子，如下：

List<Person> list = Arrays.*asList*(**new** Person(**"黄蓉"**), **new** Person(**"西门吹雪"**), **new** Person(**"欧阳峰"**));  
Person shortestName = list.get(0);  
**for** (Person p : list) {  
 **if** (p.**name**.length() < shortestName.**name**.length()) {  
 shortestName = p;  
 }  
}  
System.***out***.println(shortestName.**name**);

程序员无疑已写过成千上万次这样的for循环，其中很多都属于这个模式，下面的伪代码体现了通用模式的特点：

Object accumulator = initialValue;

for(Object element : colleciton) {

accumulator = combine(accumulator, element);

}

单词释义：accumulator：累计器，initialValue：初始值，combine：联合，结合；化合

首先赋给accumulator一个初始值：initialValue，然后在循环体中，通过调用combine函数，拿accumulator和集合中的每一 个元素做运算，再将结果赋给accumulator，最后accumulator的值 就是想要的结果。

这个模式中的两个可变项是initialValue初始值和combine函数。接下来看一 下Stream API中的reduce操作是怎么工作的。

### 3.3.7 reduce

reduce [rɪ'djuːs]：vt. 减少；降低；使处于；把…分解 vi. 减少；缩小；归纳为

reduce操作可以实现从一组值中生成一个值。前面的例子中用到的count、min、max方法，因为常用而被纳入标准库中，事实上，这些方法都是简化的reduce操作。



上图展示了如何通过reduce操作对Stream中的 数字求和。以0作为起点：一个空Stream的求和结果，每一步都将Stream中的元素累加至accumulator，遍历至Stream中的最后一个元素时，accumulator的值 就是所有元素的和。代码如下：

**int** count = Stream.*of*(1, 2, 3).reduce(0, (acc, element) -> acc + element);  
上面代码中的Lambda表达式就是reducer,它执行求和操作，有两个参数：传入Stream中的当前元素和acc，将两个参数相加，acc是累加器，保存着当前的累加结果。

reduce函数声明如下：

public abstract T reduce(T identity, BinaryOperator<T> accumulator)

函数功能为：使用提供的identity值和和关联的累积函数（BinaryOperator）对流的元素执行缩减(reducion)操作，并返回缩减结果，这相当于：

T result = identity;

for (T element : this stream)

result = accumulator.apply(result, element)

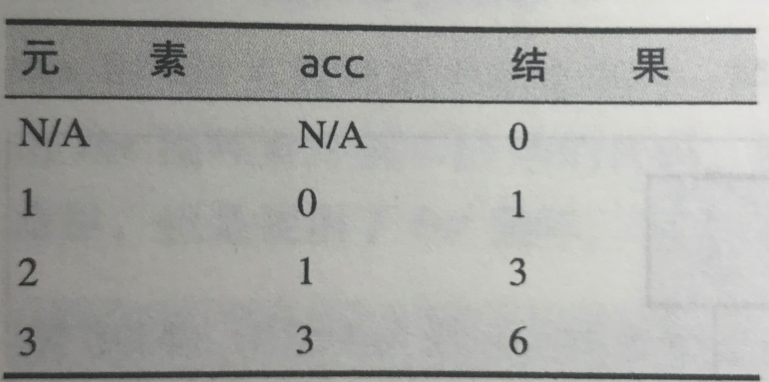
return result;

简单理解reduce的中文含义为缩减，意思应该就是说遍历吧，因为循环会越来越少至到结束。通过上面代码可知，reduce函数会遍历整个流，且我们可以传入一个identity参数，所以我们用来找最短人名也是可以的，如下：

List<Person> list = Arrays.*asList*(**new** Person(**"黄蓉"**), **new** Person(**"西门吹雪"**), **new** Person(**"欧阳峰"**));  
Person shortestNamePerson = list.get(0);  
shortestNamePerson = list.stream().reduce(shortestNamePerson, (acc, element) -> acc.**name**.length() < element.**name**.length() ? acc : element);  
System.***out***.println(shortestNamePerson.**name**);

所以，只要理解了reduce函数的原理，这个函数不仅仅是求和功能。

4.2节将介绍另外一种标准类库内置的求和方法，在实际生产环境中，应该使用那种方式，而不是使用像上面的例子中的代码方式。



这个图显示了求和过程中的中间值。实事上，可以将reduce操作展开，得到下面形式的代码：

BinaryOperator<Integer> accumulator = (acc, element) -> acc + element;  
**int** count = accumulator.apply(accumulator.apply(accumulator.apply(0, 1), 2), 3);  
System.***out***.println(count);

下面的代码可以实现同样的功能，它是命令式的Java代码，从中可以清楚看出函数式编程和命令式编程的区别：

List<Integer> numbers = Arrays.*asList*(1, 2, 3);  
**int** acc = 0;  
**for** (Integer element : numbers) {  
 acc = acc + element;  
}  
System.***out***.println(acc);

在命令式编程方式下，每一次循环将集合中 的元素和累加器相加，用相加后的结果更新累加器的值。对于集合来说，循环在外部，且需要手动更新变量。

### 3.3.8 整合操作

Stream接口的方法如此之多，有时会让人难以选择，像闯入一个迷宫，不知道该用哪个方法更好。本节将举例说明如何将问题分解为更简单的Stream操作。

第一个要解决的问题是，找出某张专辑上所有乐队的国籍。艺术家列表里既有个人，也有乐队。利用一点领域知识，假定一般乐队以定冠词The开头。当然这不是绝对的，但也差不多。

需要注意的是，这个问题不是简单的调用几个API就足以解决。这既不是使用map将一组值映射为另一组值，也不是过滤，更不是将Stream中的元素最终归约为一个值。首先，可将这 个问题分解为如下几个步骤 ：

1. 找出专辑上的所有表演者
2. 分辨出哪些表演者是乐队
3. 找出每个乐队的国籍
4. 将找出的国籍放入一个集合中

现在，找出每一步对应的Stream API就相对容易了：

1. Album类有个getMusicians方法，该方法返回一个Stream对象，包含整张专辑中所有的表演者；
2. 使用filter方法对表演者进行过虑，只保留乐队
3. 使用map方法将乐队映射为其所属国家
4. 使用collect(Collectors.toList())方法将国籍放入一个列表

最后，整合所有的操作，就得到如下代码：

Set<String> origins = album.getMusicians()

.filter(artist -> artist.getName.startWith(“The”))

.map(artist -> artist.getNationaltiy())

.collection(toSet());

这个例子将Stream的链式操作展现得淋漓尽致，调用getMusicians、filter和map方法都返回Stream对象，因此都属于惰性求值，而collect方法属于及早教求值。

现在或许是个思考的好机会，你真的需要对外暴露一个List或Set对象吗？可能一个Stream工厂才是更好的选择，通过Stream暴露集合的最大优点在于，它很好的封装了内部实现的数据结构。仅暴露一个Stream接口，用户在实际操作中无论如何使用，都不会影响内部的List或Set。

同时这也鼓励用户在编程中使用更现代的Java 8风格。不必一蹴而就，可以对已有代码渐进性地重构，保留原有的取值函数，添加返回Stream对象的函数，时间长了，就可以删掉所有返回List或Set的取值函数。清理了所有遗留代码之后，这种重构方式让人感觉棒极了！

## 3.4 重构遗留代码

为了进一步阐释如何重构遗留代码，本节将举例说明 如何将一段使用循环进行集合操作的代码，重构成基于Stream操作。重构过程中的每一步都能确保代码通过单元测试，当然你也可以自行实际操作一遍，体验并验证。

假定选定一组专辑，找出其中所有长度大于1分钟的曲目名称。下面是遗留代码：

**public** Set<String> findLongTracks(List<Album> albums) {  
 Set<String> trackNames = **new** HashSet<>();  
 **for** (Album album : albums) {  
 **for** (Track track : album.getTrackList()) {  
 **if** (track.getLength() > 60) {  
 String name = track.getName();  
 trackNames.add(name);  
 }  
 }  
 }  
 **return** trackNames;  
}

如果仔细阅读上面的这段代码，就会发现几组嵌套的循环。仅通过阅读这段代码很难看出它的编写目的，那就来重构一下（使用流来重构该代码的方式很多，下面介绍的只是其中一种。事实上，对StreamAPI越熟悉，就越不需要细分步骤。之所以在示例中一步一步地重构，完全是出于帮助大家学习的目的，在工作中无需这样做。）

第一步要修改的是for循环。首先使用Stream的forEach方法替换掉for循环，但不觉是暂时保留原来循环体中 的代码，这是在重构时非常方便的一个技巧。调用stream方法从专辑列表 中生成一个Stream，同时不要忘了getTracks方法返回的是一个Stream对象，代码如下：

**public** Set<String> findLongTracks(List<Album> albums) {  
 Set<String> trackNames = **new** HashSet<>();  
 albums.stream().forEach(album -> {  
 album.getTracks().forEach(track -> {  
 **if** (track.getLength() > 60) {  
 String name = track.getName();  
 trackNames.add(name);  
 }  
 });  
 });  
 **return** trackNames;  
}

在上面的重构中，虽然使用了流，但并没有充分发挥它的作用。实事上，重构后的代码还不如原来的代码好——天哪！因此，是时候引入一些更符合流风格的代码了，最内层的forEach方法正是主要突破口。

最内层的forEach方法有三个作用：找出长度大于1分钟的曲目，得到符合条件的曲目名称，将曲目名称加入集合。这就意味着三项Stream操作：找出满足某种条件的曲目是filter功能，得到曲目名称则可用map达成，终结操作可使用forEach方法将曲目名称加入一个集合，代码如下：

**public** Set<String> findLongTracks(List<Album> albums) {  
 Set<String> trackNames = **new** HashSet<>();  
 albums.stream().forEach(album ->  
 album.getTracks().filter(track -> track.getLength() > 60)  
 .map(track -> track.getName())  
 .forEach(name -> trackNames.add(name))  
 );  
 **return** trackNames;  
}

现在用更符合流风格人操作替换了内层的循环，但代码看起来国际机票是冗长繁琐。将各种流嵌套起来并不理想，最好还是用干净整洁的顺序调用一些方法。

理想的操作莫过于找到一种方法，将专辑转化成一个曲目的Stream。众所周知，任何时候想转化或替代代码，都该使用map操作。这里将使用比map更复杂的flatMap操作，把多个Stream合并成一个Stream并返回，代码如下：

**public** Set<String> findLongTracks(List<Album> albums) {  
 Set<String> trackNames = **new** HashSet<>();  
 albums.stream().flatMap(album -> album.getTracks())  
 .filter(track -> track.getLength() > 60)  
 .map(track -> track.getName())  
 .forEach(name -> trackNames.add(name));  
 **return** trackNames;  
}

上面的代码中使用一组简洁的方法调用替换掉两个嵌套的for循环，看起来清晰很多。然而至此并未结束，仍需要手动创建一个Set对象并将元素加入其中，但我们希望看到的是整个计算任务由一连串的Stream完成。

到目前为止，虽然还未展示转换的方法，但是已有类似 的操作，使用collecti(Collectors.toSet())可以将Stream中的值转换成一个集合。代码如下：

**public** Set<String> findLongTracks(List<Album> albums) {  
 **return** albums.stream().flatMap(album -> album.getTracks())  
 .filter(track -> track.getLength() > 60)  
 .map(track -> track.getName())  
 .collect(Collectors.*toSet*());  
}

简而言之，选取一段遗留代码进行重构，转换成使用流风格的代码。最初只是简单地使用流，但没有引入任何有用的流操作。随后通过一系列重构，最终使代码更符合使用流的风格。在上述步骤 中我们没有提到一个重点，即编写示例代码的每一步都要进行单元测试，保证代码能够正常工作，重构遗留代码时，这样做很有帮助。

## 3.5 多次调用流操作

下面两种方式都可以找出专辑上所有演出乐队的国籍：

方式一：

**public** Set<String> getOrigins(Album album) {  
 List<Artist> musicians = album.getMusicians().collect(Collectors.*toList*());  
 List<Artist> bands = musicians.stream().filter(artist -> artist.getName().startsWith(**"The"**))  
 .collect(Collectors.*toList*());  
 Set<String> origins = bands.stream().map(artist -> artist.getNationality())  
 .collect(Collectors.*toSet*());  
 **return** origins;  
}

方式二：

**public** Set<String> getOrigins(Album album) {  
 **return** album.getMusicians().filter(artist -> artist.getName().startsWith(**"The"**))  
 .map(artist -> artist.getNationality())  
 .collect(Collectors.*toSet*());  
}

用户也可以选择使用方式一来选择每一步强制对函数求值，而不是将所有方法链接在一起，但是最好不要这样做，方式一和方式二相比有如下缺点：

* 代码可读性差，样板代码太多，隐藏了真正的业务逻辑
* 效率差，每一步都要对流及是求值，生成新的集合
* 代码充斥一堆垃圾变量，它们只用来保存中间结果，除此之外毫无用处
* 难于自动并行化处理

当然，刚开始写基于流的程序时，这样的情况在所难免。但是如果发现自己经常写出这样的代码，就要反思能否将代码重构得更加简洁易读。

## 3.6 高阶函数

本章中不断出现被函数式编程的程序员称为“高阶函数”的操作。高阶函数是指接受另外一个函数作为参数，或返回一个函数的函数。高阶函数不难辨认：看函数签名就够了，如果函数的参数列表里包含函数接口，或该函数返回一个函数接口，那么该函数就是高阶函数。

Map是一个高阶函数，因为它的mapper参数是一个函数。事实上，本章介绍的Stream接口中几乎所有的函数都是高阶函数。之前的排序例子中还用到了comparing函数，它接受一个函数作为参数，获取相应的值，同时返回一个Comparator。Comparator可能会被误认为是一个对象，但它有且只有一个抽象方法，所以实际上是一个函数接口。

事实上，可以大胆断言，Comparator实际上应该是个函数，但是那时的Java只有对象，因此才造出了一个类，一个匿名类。成为对象实属巧合，函数接口向正确的方向迈出了一步。

## 3.7 正确使用Lambda表达式

本章介绍的概念能够帮助用户写出更简单的代码，因为这些概念描述了数据上的操作，明确了要达成什么转化，而不是说明如何转化。这种方式写出的代码，潜在的缺陷更少，更直接地表达了程序员的图。

明确要达成什么转化，而不是说明如何转化的另外一层含义在于写出的函数没有副作用。这一点非常重要，这样只通过函数的返回值就能充分理解函数的全部作用。

没有副作用的函数不 会改变 程序或外界的状态。本书中的第一个Lambda表达式示例是有副作用的，它向控制台输出了信息——一个可观测到的副作用。下面的代码有没有副作用？

private ActionEvent lastEvent;

private void registerHandler() {

button.addActionListener((ActionEvent event) -> {

this.lastEvent = event;

}

}

这里将参数event保存至成员变量lastEvent。给变量赋值也是一种副作用，而且更难察觉。在程序的输出中可能很难直接观察到，但是它的确更改了程序的状态。Java在这方面有局限性，例如 下面这段代码，赋值给一个局部变量localEvent:

ActionEvent localEvent = null;

Button.addActionListener(event -> localEvent = event);

这段代码试图将event赋给一个局部变量，它无法通过编译，但绝非编写错误。这实际上是语言的设计者有意为之，用以鼓励用户使用Lambda方式获取值而不是变量。获取值使用户更容易写出没有副作用的代码。如第二章所述，在Lambda表达式中使用局部变量，可以不使用final关键字，但是局部变量在即成事实上必须是final的。

无论何时，将Lambda表达式传给Stream上的高阶函数，都应该尽量避免副作用。唯一的例外是forEach方法，它是一个终结方法。

## 练习

1. 常用流操作。实现如下函数：  
   a.编写一个求和函数，计算流中所有数之和：  
      
   b.编写一个函数，接受艺术家列表作为参数，返回一个字符串列表，其中包含艺术家的姓名和国籍：  
      
    书中的答案：  
      
   c.编写一个函数，接受专辑列表作为参数，返回一个由最多包含3首歌曲的专辑组成的列表：

**public static int** addUp(Stream<Integer> numbers) {  
 **return** numbers.reduce(0, (accumulator, element) -> accumulator + element);  
}

System.***out***.println(*addUp*(Stream.*of*(1, 2, 3)));

**public static** Set<String> getArtistNameAndOrigins(List<Artist> artists) {  
 **return** artists.stream().map(artist -> artist.getName() + **" from "** + artist.getNationality())  
 .collect(Collectors.*toSet*());  
}

public static List<String> getNamesAndOrigins(List<Artist> artists) {

return artists.stream()

.flatMap(artist -> Stream.of(artist.getName(), artist.getNationality()))

.collect(toList());

}

**public static** Set<Album> getAtMostThreeTrackAlbums(List<Album> albums) {  
 **return** albums.stream().filter(album -> album.getTracks().count() <= 3).collect(Collectors.*toSet*());  
}

1. 迭代。修改如下代码，将外部迭代转换为内部迭代：  
     
     
   书中的答案如下：

**int** totalMembers = 0;  
**for** (Artist artist : artists) {  
 Stream<Artist> members = artist.getMembers();  
 totalMembers += members.count();  
}

**long** totalMembers = artists.stream().flatMap(artist -> artist.getMembers()).count();

public static int countBandMembersInternal(List<Artist> artists) {

// NB: readers haven't learnt about primitives yet, so can't use the sum() method

return artists.stream()

.map(artist -> artist.getMembers().count())

.reduce(0L, Long::sum)

.intValue();

//return (int) artists.stream().flatMap(artist -> artist.getMembers()).count();

}

1. 求值。根据Stream方法签名，判断其是惰性求值还是及早求值。  
   a. boolean anyMatch(Predicate<? Super T> predicate); 及早求值   
   b. Stream<T> limit(long maxSize); 惰性求值
2. 高阶函数。下面的Stream函数是高阶函数吗？为什么？  
   a. boolean anyMatch(Predicate<? super T> predicate); 是，因为函数接收一个函数接口作为参数  
   b. Stream<T> limit(long maxSize); 不是，因为该函数的参数即不是函数接口，且函数的返回值也不是函数接口
3. 纯函数。下面的Lambda表达式有无副作用，或者说它们是否更改了程序状态？  
   x -> x + 1  
   示例代码如下：  
   AtomicInteger count = new AtomickInteger(0);  
   List<String> origins = album.musicians().forEach(musician -> count.incAndGet();)  
   a.上述代码中传入forEach方法的Lambda表达式。  
   答：无副作用。书中的答案为：Side Effect Free
4. 计算一个字符串中小写字母的个数（提示：参阅String对象的chars方法）。  
     
   书中的答案：

**public static long** getLowerCaseCount(String string) {  
 **return** string.chars().filter(charactor -> charactor >= **'a'** && charactor <= **'z'**).count();  
}

**public static int** countLowercaseLetters(String string) {  
 **return** (**int**) string.chars().filter(Character::*isLowerCase*).count();  
}

1. 在一个字符串列表中，找出包含最多小写字母的字符串。对于空列表，返回Optional<String> 对象。  
     
   书中的答案：

**public static void** main(String[] args) **throws** Exception {  
 List<String> strings = Arrays.*asList*(**"Hello"**, **"Howabc"**, **"are"**, **"EvenDai"**);  
 System.***out***.println(*getMostestLowerCaseString*(strings).get());  
}  
  
**public static** Optional<String> getMostestLowerCaseString(List<String> strings) {  
 IntPredicate lowerCase = charactor -> charactor >= **'a'** && charactor <= **'z'**;  
 **return** strings.stream().max(Comparator.*comparing*(string -> string.chars().filter(lowerCase).count()));  
}

**public static** Optional<String> mostLowercaseString(List<String> strings) {  
 **return** strings.stream().max(Comparator.*comparing*(StringExercises::countLowercaseLetters));  
 // 这里的StringExercises是这个方法所在的类的类名  
}

## 进阶练习

1. 只用reduce和Lambda表达式写出实现Stream上的map操作的代码，如果不想返回Stream，可以返回一个List。  
   书中的答案：

import java.util.ArrayList;

import java.util.List;

import java.util.function.Function;

import java.util.stream.Stream;

public class MapUsingReduce {

public static <I, O> List<O> map(Stream<I> stream, Function<I, O> mapper) {

return stream.reduce(new ArrayList<O>(), (acc, x) -> {

// We are copying data from acc to new list instance. It is very inefficient,

// but contract of Stream.reduce method requires that accumulator function does

// not mutate its arguments.

// Stream.collect method could be used to implement more efficient mutable reduction,

// but this exercise asks to use reduce method.

List<O> newAcc = new ArrayList<>(acc);

newAcc.add(mapper.apply(x));

return newAcc;

}, (List<O> left, List<O> right) -> {

// We are copying left to new list to avoid mutating it.

List<O> newLeft = new ArrayList<>(left);

newLeft.addAll(right);

return newLeft;

});

}

}

1. 只用reduce和Lambda表达式写出实现Stream上的filter操作的代码，如果不想返回Stream，可以返回一个List。  
   书中的答案：

import java.util.ArrayList;

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Stream;

public class FilterUsingReduce {

public static <I> List<I> filter(Stream<I> stream, Predicate<I> predicate) {

List<I> initial = new ArrayList<>();

return stream.reduce(initial,

(List<I> acc, I x) -> {

if (predicate.test(x)) {

// We are copying data from acc to new list instance. It is very inefficient,

// but contract of Stream.reduce method requires that accumulator function does

// not mutate its arguments.

// Stream.collect method could be used to implement more efficient mutable reduction,

// but this exercise asks to use reduce method explicitly.

List<I> newAcc = new ArrayList<>(acc);

newAcc.add(x);

return newAcc;

} else {

return acc;

}

},

FilterUsingReduce::combineLists);

}

private static <I> List<I> combineLists(List<I> left, List<I> right) {

// We are copying left to new list to avoid mutating it.

List<I> newLeft = new ArrayList<>(left);

newLeft.addAll(right);

return newLeft;

}

}

# 类库

前3章讨论了如何编写Lambda表达式，接下来将详细阐述另一 佧重要方面：如何使用Lambda表达式。即使不 需要编写像Stream这样重度使用函数式编程风格的类库，学会如何使用Lamb表达式也是非常重要的。即使一个最简单的应用，也可能因为代码即数据的函数式编程风格而受益。

Java 8中的另一个变化是引入了默认方法和 接口的静态方法，它改变了人们认识类库的方式，接口中的方法也可以包含代码体了。

本章还对前3章疏漏的知识 点进行补充，比如 ，Lambda表达式方法重载的工作原理、基本类型的使用方法等。使用Lambda表达式编写程序时，掌握这些知识 非常重要。

## 4.1 在代码中使用Lambda表达式

让我们来看一个日志系统中的具体案例。在slf4j和log4j等几种常用的日志系统中，有一些记录日志的方法，当日志级别不低于某个固定级别时就会开始记录日志。如此一来，在日志框架中设置类似void debug(String message)这样的方法，当级别为debug时，它们就开始记录日志消息。

问题在于，频繁计算消息是否应该记录日志会对系统性能产生影响。程序员通过显示调用isDebugEnabled方法来优化系统性能，如下面代码所示，即使直接调用debug方法能省去记录文本信息，也仍然需要调用expensiveOperation方法，并且需要将执行结果和已有字符串连接起来，因此，使用if语句显式判断，可以让程序跑得更快。

Logger logger = new Logger();

If (logger.isDebugEnable()) {

Logger.debug(“Look at this: “ + expensiveOperation());

}

这里我们想做的是传入一个Lambda表达式，生成一条用作日志信息的字符串。只有日志级别在调试或以上级别时，才会执行该Lambda表达式。使用这个方式重写上面的 代码，如下：

Logger logger = new Logger();

Logger.debug(() -> “Look at this: “ + expensiveOperation());

那么在Logger类中该方法是如何实现的呢？从类库的角度，我们可以使用内置的Supplier函数接口，它只有一个get方法。然后通过调用isDebugEnabled判断是 否需要记录日志，是否需要调用get方法，如果需要，就调用get方法并将结果传给debug方法，如下：

Public void debug(Supplier<String> message) {

If (isDebugEnabled()) {

debug(message.get());

}

}

调用get方法，相当于传入的Lambda表达式。这种方式也能和匿名内部类一起工作，如果用户暂时无法升级到Java8，这种方式可以实现向后兼容。

值得注意的事，不 同的函数接口有不同的方法。如果使用Predicate，就应该调用test方法，如果使用Function，就应该调用apply方法。

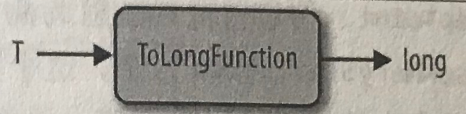
## 基本类型

以上部分还没有用到基本类型。在Java中，有一些相伴的类型，比如int和Integer，前者是基本类型，后者是装箱类型。基本类型内建在语言和运行环境中，是基本的程序构建模块；而装箱类型属于普通的Java类，只不过是对基本类型的一种封装。

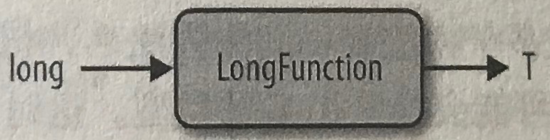
Java的泛型是基于对泛型参数类型的擦除——换句话说，假设它是Object对象的实例——因此只有装箱类型才能作为泛型参数。这就解释了为什么在Java中想要一个包含整型值的列表List<int>，实际上得到的却是一个包含整型对象的列表List<Integer>。

麻烦的 是，由于装箱类型是对象，因此在内存中存在额外的开销。比如，整型在内存中占有用4字节，整型对象却要占用16字节。这一情况在数组上更加严重，整形数组中的每个元素只占用基本类型的内存，而整型对象数组中，每个元素都是内存中的一个指针，指向Java堆中的某个对象。在最坏的情况下，同样大小的数组，Integer[] 要比int[]多占用6倍内存。

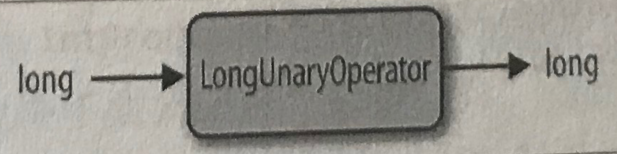
将基本类型转换为装箱类型，称为装箱，反之则称为拆箱，两者都需要额外的计算开销。对于需要大量数值运算的算法来说，装箱和拆箱的计算开销，以 及装箱类型占用的额外内存，会明显减缓程序的运行速度。



为了减小这些性能开销，Stream类的某些方法对基本类型和装箱类型做了区分，如上图所示显示的高阶函数mapToLong和其他类似函数即为该方面的一个尝试。在Java8中，仅对下整型、长整 型和双浮点型做了特殊处理，因为它们在数值计算中用得最多，特殊处理后的系统性能提升效果最明显。



对基本类型做特殊处理的方法命名上有明确的规范。如果方法返回类型为基本类型，则在基本类型前加To，如ToLongFunction。如果参数是基本类型，则不加前缀只需要类型名称即可，如LongFunction。如果高阶函数使用基本类型，则在操作后加后缀to再加基本类型，如mapToLong。



这些基本类型都有与之对应的Stream，以基本类型名为前缀，如LongStream。事实上，mapToLong方法返回的不是一个一般的Stream，而是一个特殊处理的Stream。在这个特殊的Stream中，map方法的 实现方式也不 同，它接受 一个LongUnaryOperator函数，将一个长整型值映射成另一个长整型值，如上图所示。通过一些高阶函数装箱方法，如mapToObj，也可以从一个基本类型的Stream得到一个装箱后的Stream，如Stream<Long>。

如有可能 ，应尽可能 多地使用对基本类型做过特殊处理的方法，进而改善性能。这些特殊的Stream还提供额外的方法，避免重复实现一些通用的方法，让代码更能体现出数值计算的意图。

下面代码展示了IntStream的使用：

Statistics [stə'tɪstɪks]：n. 统计

Summary ['sʌm(ə)rɪ]：adj. 简易的；扼要的 n. 概要，摘要，总结

**public static void** printTrackLengthStatistics(Album album) {  
 IntSummaryStatistics trackLengthStats = album.getTracks().mapToInt(track -> track.getLength())  
 .summaryStatistics();  
 System.***out***.printf(**"Max: %d, Min: %d, Ave: %f, Sum: %d"**,  
 trackLengthStats.getMax(),  
 trackLengthStats.getMin(),  
 trackLengthStats.getAverage(),  
 trackLengthStats.getSum());  
}

这里使用对基本类型特殊处理的方法mapToInt，将Stream<Track>映射成IntStream,IntStream包含一个summaryStatistics方法。这个方法在DoubleStream、LongStream等基本类型包装的Stream类中都有。

## 重载解析

在Java中可以重载方法，即方法名一样，而方法的参加不一样，这在推断参数类型时会带来问题，因为系统可能推断出多种类型。这时，javac会挑出最具体的类型。如下，输出String，而不是Object。

**public static void** overloadedMethod(Object o) {  
 System.***out***.println(**"Object"**);  
}  
**public static void** overloadedMethod(String s) {  
 System.***out***.println(**"String"**);  
}  
**public static void** main(String[] args) **throws** Exception {  
 *overloadedMethod*(**"ha ha"**); *// 输出String*  
}

BinaryOperator是一种特殊的Bifunction类型，参数的类型和返回值类型相同。比如两佧整数相加就是一个BinaryOperator.

Lambda表达式的类型就是对应的函数接口类型，因此，将Lambda表达式作为参数传递时，情况也依然如此。操作时可以重载一个方法，分别接受BinaryOperator和该抱头痛哭中的一个子接口作为参数。调用这些方法时，Java推导出的Lambda表达式的类型是最具体的函数接口类型，示例如下：

**public interface** IntegerBiFunction **extends** BinaryOperator<Integer> { }  
**public static void** overloadedMethod(BinaryOperator<Integer> Lambda) {  
 System.***out***.println(**"BinaryOperator"**);  
}  
**public static void** overloadedMethod(IntegerBiFunction Lambda) {  
 System.***out***.println(**"IntegerBiFunction"**);  
}  
**public static void** main(String[] args) **throws** Exception {  
 *overloadedMethod*((x, y) -> x + y); *// 输出IntegerBiFunction*}

当然，同时存在多个重载方法时，哪个是“最具体的类型”可能并不能明确，如下所示：

**public interface** IntPredicate {  
 **public boolean** test(**int** value);  
}  
**public static void** overloadedMethod(Predicate<Integer> predicate) {  
 System.***out***.println(**"Predicate"**);  
}  
**public static void** overloadedMethod(IntPredicate predicate) {  
 System.***out***.println(**"IntPredicate"**);  
}  
**public static void** main(String[] args) **throws** Exception {  
 overloadedMethod(x -> **true**); *// 编译不通过，提示模糊的方法调用  
 overloadedMethod*((IntPredicate) x -> **true**); *// 编译通过  
 overloadedMethod*((Predicate) x -> **true**); *// 编译通过*}

传入overloadedMethod方法的Lambda表达式和两个函数接口Predicate、IntPredicate在类型上都是匹配的 。在这段代码中，两种情况都定义了相应的重载方法，这时，javac就无法编译，在错误报告中显示Lambda表达式被模糊调用。IntPredicate没有继承Predicate，因此编译器无法推荐出哪个类型更具体。

将Lambda表达式强制转换为IntPredicate或Predicate<Integer>类型可以解决这个问题，至于转换为哪种类型取决于要调用哪个函数接口。当然，如果以前你曾自行设计过类库，就可以将其视为“代码异味”，不该再重载，而应当开始重新命名重载方法。

## @FunctionalInterface

* functional['fʌŋkʃənəl] adj. 功能的

FunctionnalInterface翻译过来就是“函数接口”的意思。

虽然2.4节已讨论过函数接口的定义标准，但未提示@FunctionalInterface注释。事实上，每个用作函数接口的接口都应该添加这个注释。

这究竟是什么意思呢?Java中有一些接口，虽然只含一个方法，但并不是为了使用Lambda表达式来实现的。比如，有些对象内部可能保存着某种状态，使用带有一个方法的接口可能纯属巧合。java.lang.Comparable和java.io.Closeable就属于这样的情况。

如果一个类是可比较的，就意味着在该类的实例之间存在某种顺序，比如字符串中的字母顺序。人们通常不会认为函数是可比较的，如果一个东西既没有属性也没有状态，拿什么比较呢?

一个可关闭的对象必须持有某种打开的资源，比如一个需要关闭的文件句柄。同样，该接口也不能是一个纯函数，因为关闭资源是更改状态的另一种形式。

和Closeable和Comparable接口不同，为了提高Stream对象可操作性而引入的各种新接口，都需要有Lambda表达式可以实现它。它们存在的意义在于将代码块作为数据打包起来。因此，它们都添加了@Functionallnterface注释。

该注释会强制javac检査一个接口是否符合函数接口的标准。如果该注释添加给一个枚举类型、类或另一个注释，或者接口包含不止一个抽象方法，javac就会报错。重构代码时，使用它能很容易发现问题。

## 二进制接口的兼容性

如第3章开篇所言，Java 8中对API最大的改变在于集合类。虽然Java在持续演进，但它一直在保持着向后二进制兼容。具体来说，使用Java 1到Java 7编译的类库或应用，可以直接在Java 8上运行。

当然，错误也难免会时有发生，但和其他编程平台相比，二进制兼容性一直被视为Java的关健优势所在。除非引人新的关键字，如enum，达成源代码向后兼容也不是没有可能实现。可以保证，只要是Java 1到Java 7写出的代码，在Java 8中依然可以编译通过。

事实上，修改了像集合类这样的核心类库之后，这一保证也很难实现。我们可以用具体的例子作为思考练习。Java 8中为Coltection接口增加了stream方法，这意味着所有实现了Collection接口的类都必须增加这个新方法。对核心类库里的类来说，实现这个新方法(比如为ArrayList增加新的stream方法)就能使问题迎刃而解。

缺憾在于，这个修改依然打破了二进制兼容性，在JDK之外实现Collection接口的类，例如MyCustomList，也仍然需要实现新增的stream方法。这个MyCustomList在Java 8中无法通过编译，即使已有一个编译好的版本，在JVM加载MyCustomList类时，类加载器仍然会引发异常。

这是所有使用第三方集合类库的梦魇，要避免这个糟糕情况，则需要在Java 8中添加新的语言特性：默认方法

## 默认方法

Collection接口中增加了新的stream方法，如何能让MyCustomList类在不知道该方法的情况下通过编译？Java 8通过如下方法解决该问题：Collection接口告诉它所有的子类：“如果你没有实现stream方法，就使用我的吧。”接口中这样的方法叫作默认方法，在任何接口中，无论函数接口还是非函数接口，都可以使用该方法。

Iterable接口中也新增了一个默认方法forEach，该方法功能和for循环类似，但是允许用户使用一个Lambda表达式作为循环体。如下示例展示了JDK中forEach的实现方式：

default void forEach(Consumer<? super T> action) {

for (T t : this) {

action. accept (t);

}

}

* Consumer  [kən'sjuːmə] n. 消费者；用户，顾客

如果已经习惯了通过调用接口方法来使用Lambda表达式的方式，那么这个例子理解起来就相当简单。它使用一个常规的for循环遍历Iterable对象，然后对每个值调用accept方法。

既然如此简单，为何还要单独提出来呢?重点就在于代码段前面的新关键字default。这个关键字告诉javac用户真正需要的是为接口添加一个新方法。除了添加了一个新的关键字，默认方法在继承规则上和普通方法也略有区别。

和类不同，接口没有成员变量，因此默认方法只能通过调用子类的方法来修改子类本身，避免了对子类的实现做出各种假设。

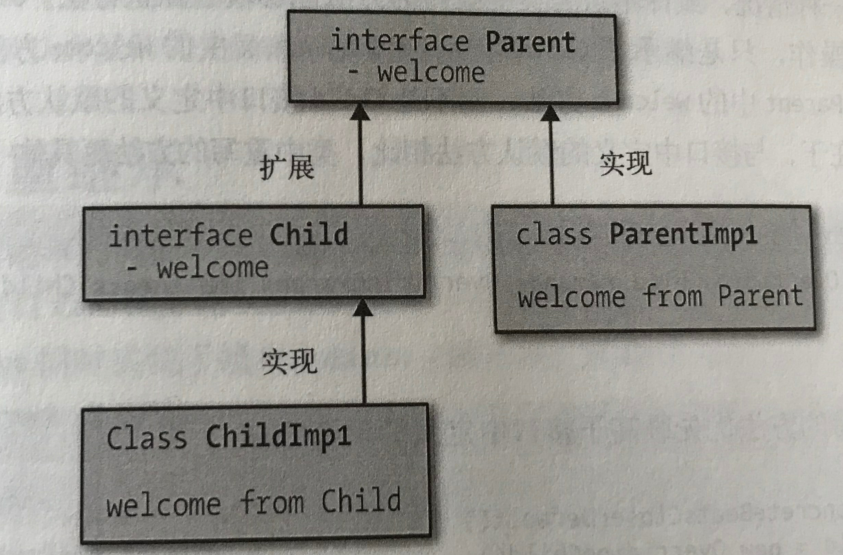
**默认方法和子类**

**public interface** Parent {  
 **default void** welcom() {  
 System.***out***.println(**"Parent"**);  
 }  
}  
  
*// 此类并没有实现接口中的welcom方法***public static class** ParentImpl **implements** Parent { }  
  
**public static void** main(String[] args) **throws** Exception {  
 Parent p = **new** ParentImpl();  
 p.welcom();  
}

此时可新建一个Child接口，继承自Parent接口，Child实现了自己的默认welcom方法，如下：

**public interface** Child **extends** Parent {  
 @Override  
 **default void** welcom() {  
 System.***out***.println(**"Child"**);  
 }  
}  
  
**public static class** ChildImpl **implements** Child { }  
  
**public static void** main(String[] args) **throws** Exception {  
 Parent p = **new** ChildImpl();  
 p.welcom(); // 输出Child  
}

继承关系图如下：



实验证明，默认方法也可以被覆盖。

且当extands与implements同时存在时，如果有相同的默认方法，则使用extands中的，因为类中重写方法更具体，示例如下：

**public static class** ParentImpl **implements** Parent {  
 @Override  
 **public void** welcom() {  
 System.***out***.println(**"ParentImpl"**);  
 }  
}  
  
**public static class** ChildImpl **extends** ParentImpl **implements** Child {}  
  
**public static void** main(String[] args) **throws** Exception {  
 Parent p = **new** ChildImpl();  
 p.welcom(); *// 输出ParentImpl*}

简言之，类中重写的方法胜出。这样的设计主要是由增加默认方法的 目的决定的，增加默认方法主要是为了在接口上向后兼容，让类中重写方法的优先级高于默认方法能简化很多继承问题。假设已实现了一 个定制的列表 MyCustomList，该类中有一个addAll方法，如果新的List接口中也增加了一个默认方法addAll，该方法将列表的操作代理到add方法。如果类中 重写的方法没有默认方法的优先级别高，那么 就会破坏已有的实现。

## 4.7 多重继承

**public interface** A {  
 **default void** func() {  
 System.***out***.println(**"A"**);  
 }  
}  
  
**public interface** B {  
 **default void** func() {  
 System.***out***.println(**"B"**);  
 }  
}  
  
**public interface** C **extends** A, B {}

此时，javac并不明确应该继承哪个接口中的func方法，因此编译器会报错：class C inherits unrelated defaults for func() from types A and B。这没办法知道要继承哪个接口上的默认方法，所以报错，如果是类去实现则不 会有这个问题，如下：

**public class** C **implements** A, B {  
 @Override  
 **public void** func() {  
 A.**super**.func();  
 }  
}

这里使用了增加的super语法，用来指明使用接口A中定义的默认方法。此前，使用super关键字是指向父类，现在使用类似InterfaceName.superp 这样的语法指的是继承自父接口的方法。

如果对默认方法的工作原理，特别是在多重继承下的行为还没有把握，如下三条简单的规律可以帮助大家。

1. 类胜于接口。如果在继承链中有方法体或抽象的方法声明，那么就可以忽略接口中定义的方法。
2. 子类胜于父类。如果一个接口继承了另一个接口，且两个接口都定义了一个默认方法，那么子类中定义的方法胜出。
3. 没有规则三。如果上面两条规则不适用，子类要么需要实现该方法，要么将该方法声明

其中第一条规则是为了让代码向后兼容。

## 4.8权衡

在接口中定义方法的诸多变化引发了一系列问题，既然可用代码主体定义方法，那Java 8中的接口还是旧有版本中界定的代码吗?现在的接口提供了某种形式上的多重继承功能，然而多重继承在以前饱受诟病，Java因此舍弃了该语言特性，这也正是Java在易用性方面优于C+十的原因之一。

语言特性的利弊也在不断演化。很多人认为多重继承的问题在于对象状态的继承，而不是代码块的继承，默认方法避免了状态的继承，也因此避免了C++中多重继承的最大缺点。

突破语言上的局限性吸引着无数优秀的程序员不断尝试。现在已有一些博客文章，阐述在Java 8中实现完全的多重继承做出的尝试，包括状态的继承和默认方法。尝试突破Java 8这些有意为之的语言限制时，却往往又掉进C++的旧有陷阱之中。

接口和抽象类之间还是存在明显的区别。接口允许多重继承，却没有成员变量；抽象类可以继承成员变量，却不能多重继承。在对问题域建模时，需要根据具体情况进行权衡，而在以前的Java中可能并不需要这样。

## 4.9接口的静态方法

前面已多次出现过stream.of方法的调用，接下来将对其进行详细介绍。stream是个接口，Stream.of是接口的静态方法。这也是Java 8中添加的一个新的语言特性，旨在帮助编写类库的开发人员，但对于日常应用程序的开发人员也同样适用。

人们在编程过程中积累了这样一条经验，那就是一个包含很多静态方法的类。有时，类是一个放置工具方法的好地方，比如Java 7中引人的Objects类，就包含了很多工具方法，这些方法不是具体属于某个类的。

当然，如果一个方法有充分的语义原因和某个概念相关，那么就应该将该方法和相关的类或接口放在一起，而不是放到另一个工具类中。这有助于更好地组织代码，阅读代码的人也更容易找到相关方法。

比如，如果想创建一个由简单值组成的stream，自然希望stream中能有一个这样的方法。这在以前很难达成，引人重接口的stream对象，最后促使Java为接口加入了静态方法。

Stream和其他几个子类还包含另外几个静态方法。特别是range和iterate方法提供了产生Stream的其他方式。

## 4.10 Optional

* optional  ['ɑpʃənl] adj. 可选择的，随意的 n. 选修科目
* reduce  [rɪ'djuːs] vt. 减少；降低；使处于；把…分解 vi. 减少；缩小；归纳为

reduce方法的一个重点尚未提及：reduce方法有两种形式，一种如前面出现的需要有一个初始值，另一种变式则不需要有初始值。没有初始值的情况下，reduce的第一步使用stream中的前两个元素。有时，reduce操作不存在有意义的初始值，这样做就是有意义的，此时，reduce方法返回一个Optional对象。

Optional是为核心类库新设计的一个数据类型，用来替换null值。人们对原有的nuLL值有很多抱怨，甚至连发明这一概念的Tony Hoare也是如此，他曾说这是自己的一个“价值连城的错误”。作为一名有影响力的计算机科学家就是这样：虽然连一毛钱也见不到，却也可以犯一个“价值连城的错误”。

人们常常使用null值表示值不存在，Optional对象能更好地表达这个概念。使用null代表“值不存在”的最大问题在于NullPointerException。一且引用一个存储null值的变量工，程序会立即崩溃。使用Optional对象有两个目的：首先，Optional对象鼓励程序员适时检査变量是否为空，以避免代码缺陷; 其次，它将一个类的API中可能为空的值文档化，这比阅读实现代码要简单很多。

下面我们举例说明Optional对象的API,从而切身体会一下它的使用方法。使用工厂方法of，可以从某个值创建出一个Optional对象。Optionat对象相当于值的容器，而该值可以通过get方法提取。如下：

Optional<String> opt = Optional.*of*(**"hello"**);  
System.***out***.println(opt.get()); *// 输出hello*

Optional对象也可能 为空，因此还有一个对应的工厂方法empty，另外一个工厂方法ofNullabe则可以将一个空值转换成Optional对象。isPresent方法用于表示Optional对象里是中有值。示例如下：

Optional emptyOpt1 = Optional.*empty*();  
Optional emptyOpt2 = Optional.*ofNullable*(**null**);  
System.***out***.println(emptyOpt1.isPresent()); *// false*System.***out***.println(emptyOpt2.isPresent()); *// false*System.***out***.println(Optional.*of*(**"hello"**).isPresent()); *// true*System.***out***.println(Optional.*ofNullable*(**"hello"**).isPresent());*// true*

使用Optional对象的方式之一是在调用get()方法前，先使用isPresent检查Optional对象是否有值。使用orElse方法则更简洁，当Optional对象为空时，该方法提供了一个备选值。如果计算备选值在计算上太过繁琐，即可使用orElseGet方法。该方法接受一个Supplier对象，只有在Optional对象真正为空时才会调用，如下展示了这两个方法：

Optional<String> emptyOpt = Optional.*empty*();  
String result1 = emptyOpt.orElse(**"b"**);  
String result2 = emptyOpt.orElseGet(() -> **"c"**);  
System.***out***.println(result1 + **", "** + result2); *// 输出：b, c*

orElse和orElseGet都是 在Optional中无值时提供一个备选值，orElse是直接就提供一个备选值，而orElseGet因为接收的参数是一个Lambda，所以在返回一个备选值前，可以在Lambda表达式中进行一些计算或判断什么的。

Optional对象不 仅可以用于新的Java 8 API，也可以用于具体领域类中，和普通的类别无二致。当试图避免空值相关的缺陷，如未捕获的异常时，可以考虑一下是否可使用Optional对象。

## 4.11 要点回顾

* 便用基本类型定制的Lambda表达式和Stream，如IntStream可以显著提升系统性能。
* 默认方法是指接口中定义的包含方法体的方法，方法名有defauIt关键字做前缀。
* 在一个值可能为空的建模情况下，使用Optional对象能替代使用null值

## 4.12 练习

1.在下面例子的Performance接口基础上，添加getAllMusicians方法，该方法返回包含所有艺术家名字的stream，如果对象是乐队，则返回每个乐队成员的名字。例如，如果getMusicians方法返回甲壳虫乐队，则getAllMusicians方法返回乐队名和乐队成员，如约翰·列位、保罗·麦卡特尼等。

/\* 该接口表示艺术家的演出——专辑或演唱会 \*/

public interface Perfornance {

public String getName();

public Stream<Artist> getMusicians();

}

答案：

public default Stream<Artist> getAllMusicians() {

return getMusicians().flatMap(artist -> concat(Stream.of(artist), artist.getMembers()));

}

2.根据前面描述的重载解析规则，能否重写默认方法中的 equals或hashCode方法？  
答：No - they are defined on java.lang.Object, and 'class always wins.' 翻译结果：不行，它们是在java.lang.Object上定义的，并且类总是赢家。 因为这两个方法在Object上有，如果一个接口定义有这两个方法为默认方法，则这两个方法是多余的，因为用不上，因为一个类默认会继承Object的，而类中定义的方法优先级高，会继承到Object中的equals和hashCode方法。

3.下面的示例中的 Artists类表示一组艺术家，重构该类，使得getArtist方法返回一个Optional<Artist>对象。如果索引在有效范围内，返回对应的元素，否则返回一个空Optional对象。此外，还需要重构getArtistName方法，保持相同的行为。

public class Artists {

private List<Artist> artists;

public Artists(List<Artist> artists) {

this.artists = artists;

}

public Artist getArtist(int index) {

if (index < 0 || index >= artists.size()) {

indexException(index);

}

return artists.get(index);

}

private void indexException(int index) {

throw new IllegalArgumentException(index + " doesn't correspond to an Artist");

}

public String getArtistName(int index) {

try {

Artist artist = getArtist(index);

return artist.getName();

} catch (IllegalArgumentException e) {

return "unknown";

}

}

}

答案如下：

public Optional<Artist> getArtist(int index) {

if (index < 0 || index >= artists.size()) {

return Optional.empty();;

}

return Optional.of (artists.get(index));

}

public Aptional<String> getArtistName(int index) {

try {

Artist artist = getArtist(index);

return Optional.ofNullable(artist.getName());

} catch (IllegalArgumentException e) {

return Optional.of("unknown");

}

}

书中的答案：

public String getArtistName(int index) {

Optional<Artist> artist = getArtist(index);

return artist.map(Artist::getName).orElse("unknown");

}

## 4.13 开放练习

审阅工作代码库颧熟悉的开源项目代码，找出哪些只包含静态方法的类适合用包含静态方法的接口替代。如有可能，和同事一起讨论，看他们是否赞同你找出的结果。

# 第5章 高级集合类和收集器

第3章只介绍了集合类的部分变化，事实上，Java 8对集合类的改进不止这些。现在是时候介绍一些高级主题了，包括新引入的Collector类。同时我还会为大家介绍方法引用，它可以帮助大家在Lambda表达式中轻松使用已有代码。编写大量使用集合类的代码时，使用方法引用能让程序员获得丰厚的回报。本章还会涉及集合类的一些更高级的主题，比如流中元素的顺序，以及一些有用的API.

## 5.1方法引用.

读者可能已经发现，Lambda表达式有一个常见的用法Lambda表达式经常调用参数。比如想得到艺术家的姓名，Lambda的表达式如下

artist -> artist.getName();

这种用法如此普遍，因此Java 8为其提供了一个简写语法，叫作方法引用，帮助程序员重用已有方法。用方法引用重写上面的Lambda表达式，代码如下

Artist::getNane

标准语法为Classnane::methodNane。需要注意的是，虽然这是一个方法，但不需要在后面加括号，因为这里并不调用该方法。我们只是提供了和Lambda表达式等价的一种结构，在需要时才会调用。凡是使用Lambda表达式的地方，就可以使用方法引用。

构造函数也有同样的缩写形式，如果你想使用Lambda表达式创建一个Artitst对象，可能会写出如下代码：

(name, nationality) -> new Artist(name, nationality)

使用方法引用，上述代码可写为：

Artist::new

这段代码不仅比原来的代码短，而且更易阅读。Artist::new立刻告诉程序员这是在创建一个Artist对象,程序员无需看完整行代码就能弄明白代码的意图。另一个要注意的地方是方法引用自动支持多个参数，前提是选对了正确的函数接口。

还可以用这种方式创建数组，下面的代码创建了一个字符串型的数组

String[]::new

从现在开始，我们将在合适的地方使用方法引用，因此读者很快会看到更多的例子。一开始探索Java 8时，有位朋友告诉我，方法引用看起来“就像在作弊”。他的意思是说，了解如何使用Lambda表达式让代码像数据一样在对象间传递之后，这种直接引用方法的方式就像“作弊”。

放心，这不是在作弊。读者只要记住，每次写出形如x -> foo(x)的Lambda表达式时，和直接调用方法foo是一样的。方法引用只不过是基于这样的事实，提供了一种简短的语法而已。

## 5.2元素顺序

另外一个尚未提及的关于集合类的内容是流中的元素以何种顺序排列。读者可能知道，一些集合类型中的元素是按顺序排列的，比如List；而另一些则是无序的，比如Hashset。增加了流操作后，顺序问题变得更加复杂。

直观上看，流是有序的，因为流中的元素都是按顺序处理的。这种顺序称为出现顺序。出现顺序的定义依赖于数据源和对流的操作。

在一个有序集合中创建一个流时，流中的元素就按出现顺序排列，因此，例5-1中的代码

总是可以通过。

例5-1顺序测试永远通过：

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4);  
List<Integer> sameOrder = numbers.stream().collect(Collectors.*toList*());  
System.***out***.println(numbers.equals(sameOrder)); *// 结果总是true*

如果集合本身就是无序的，由此生成的流也是无序的。Hashset就是一种无序的集合,此不能保证例5-2所示的程序每次都通过。

Set<Integer> numbers = **new** HashSet<>(Arrays.*asList*(1, 2, 3, 4));  
List<Integer> list1 = numbers.stream().collect(Collectors.*toList*());  
List<Integer> list2 = Arrays.*asList*(1, 2, 3, 4);  
System.***out***.println(list1.equals(list2)); *// 结果有时候为false*

流的目的不仅是在集合类之间转换，而且同时提供了 一组处理数据的通用操作。有些集合本身是无序的，但这些操作有时会产生顺序，试看的代码：

Set<Integer> numbers = **new** HashSet<>(Arrays.*asList*(4, 1, 3, 2));  
List<Integer> list1 = numbers.stream().sorted().collect(Collectors.*toList*());  
List<Integer> list2 = Arrays.*asList*(1, 2, 3, 4);  
System.***out***.println(list1.equals(list2)); *// 结果为true*

一些中间操作会产生顺序，比如对值做映射时，映射后的值是有序的，这种顺序就会保留下来。如果进来的流是无序的，出去的流也是无序的。看一下下面的例子，我们只能说HashSet中含有元素，但对其顺序不能做出任何假设，因为HashSet是无序的，使用了映射操作后，得到的集合仍然是无序的。

本例中关于顺序的假设是永远正确的：

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4);  
List<Integer> stillOrdered = numbers.stream().map(x -> x + 1).collect(Collectors.*toList*());  
  
*// 顺序得到了保留*System.***out***.println(Arrays.*asList*(2, 3, 4, 5).equals(stillOrdered));  
  
Set<Integer> unordered = **new** HashSet<>(numbers);  
List<Integer> stillUnordered = unordered.stream().map(x -> x + 1).collect(Collectors.*toList*());  
  
*// 顺序得不到保证*System.***out***.println(stillUnordered.contains(2));  
System.***out***.println(stillUnordered.contains(3));  
System.***out***.println(stillUnordered.contains(4));  
System.***out***.println(stillUnordered.contains(5));

一些操作在有序的流上开销更大，调用unordered方法消除这种顺序就能解决该问题。大多数操作都是在有序流上效率更高，比如filter、map和reduce等。

这会带来一些意想不到的结果，比如使用并行流时，forEach方法不能保证元素是按顺序处理的（第6章会详细讨论这些内容）。如果需要保证按顺序处理，应该使用forEachOrdered方法，它是你的朋友。

## 5.3 使用收集器

前面我们使用过collect(toListo())，在流中生成列表。显然，List是能想到的从流中生成的最自然的数据结构，但是有时人们还希望从流生成其他值，比如Map或Set，或者你希望定制一个类将你想要的东西抽象出来。

前面已经讲过，仅凭流上方法的签名，就能判断出这是否是一个及早求值的操作。reduce操作就是一个很好的例子，但有时人们希望能做得更多。

这就是收集器，一种通用的、从流生成复杂值的结构。只要将它传给collect方法，所有的流就都可以使用它了。

标准类库已经提供了一些有用的收集器，让我们先来看看。本章示例代码中的收集器都是从java.util.stream.Correctors类中静态导人的。

### 5.3.1 转换成其他集合

有一些收集器可以生成其他集合。比如前面已经见过的toList，生成了java.util.List类的实例。还有toSet和toCollection，分别生成Set和Collection类的实例。到目前为止，我已经讲了很多流上的链式操作，但总有一些时候，需要最终生成一个集合——比如:

* 已有代码是为集合编写的，因此需要将流转换成集合传入
* 在集合上进行一系列链式操作后，最终希望生成一个值
* 写单元测试时，需要对某个具体的集合做断言。

通常情况下，创建集合时需要调用适当的构造函数指明集合的具体类型:

List<Artist> artists = new ArrayList<>();

但是调用toList或者toSet方法时，不需要指定具体的类型。stream类库在背后自动为你挑选出了合适的类型，本书后面会讲述如何使用stream类库并行处理数据，收集并行操作的结果需要的Set，和对线程安全没有要求的Set类是完全不同的。

可能还会有这样的情况，你希望使用一个特定的集合收集值，而且你可以稍后指定该集合的类型，比如，你可能希望地使用TreeSet，而不是由框架在背后自动为你指定一种类型的Set。此时就可以使用toCollection，它接受一个函数作为参数，来创建集合，示例如下：

Stream.collect(toCollection(TreeSet::new));

### 5.3.2 转换成值

还可以利用收集器让流生成一个值。maxBy和minBy允许用户按某种特定的顺序生成值。例5-6展示了如何找出成员最多的乐队。它使用一个Lambda表达式，将艺术家映身为成员数量，然后定义了一个比较器，并将比较器传入maxBy收集器。

例5-6找出成员最多的乐队

**public** Optional<Artist> biggestGroup(Stream<Artist> artists) {  
 Function<Artist, Long> getCount = artist-> artist.getMembers().count();  
 **return** artists. collect(*maxBy*(Comparator.*comparing*(getCount)));  
}

ninBy就如它的方法名，是用来找出最小值的。

还有些收集器实现了一些常用的数值运算。让我们通过一个计算专辑曲目平均数的例子来看看，如例5-7所示。

例5-7找出一组专辑上曲目的平均数：

**public double** averageNumberOfTracks(List<Album> albums) {  
 **return** albums.stream().collect(*averagingInt*(album -> album.getTrackList().size()));  
}

和以前一样，通过调用stream方法让集合生成流，然后调用collect方法收集结果。averaginglnt方法接受一个Lambda表达式作参数，将流中的元素转换成一个整数，然后再计算平均数。还有和double和long类型对应的重载方法，帮助程序员将元素转换成相应类型的值。

第4章介绍过一些特殊的流，如IntStream，为数值运算定义了一些额外的方法。事实上，Java 8也提供了能完成类似功能的收集器，如averaginglnt。可以使用sumingInt及其重载方法求和。Summarystatistics也可以使用summinglnt及其组合收集。

### 5.3.3数据分块

另外一个常用的流操作是将其分解成两个集合。假设有一个艺术家组成的流，你可能希望将其分成两个部分，一部分是独唱歌手，另一部分是由多人组成的乐队。可以使用两次过滤操作，分别过滤出上述两种艺术家。

但是这样操作起来有问题。首先，为了执行两次过滤操作，需要有两个流。其次，如果过滤操作复杂，每个流上都要执行这样的操作，代码也会变得冗余。

幸好我们有这样一个收集器partitionngBy，它接受一个流，并将其分成两部分(如图5-1所示)。它使用Predicate对象判断一个元素应该属于哪个部分，并根据布尔值返回一个Map到列表。因此，对于true List中的元素，Predicate返回true；对其他List中的元素，Predicate返回false。

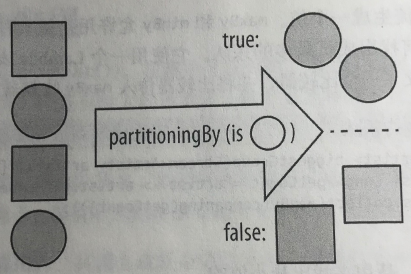


图5-1：partitioninBy收集器

使用它，我们就可以将乐队（有多个成员）和独唱歌手分开了。在本例中，分块函数指明艺术家是否为独唱歌手。实现如例5-8所示。

例5- 8 将艺术家组成的流分成乐队和独唱歌手两部分:

**public** Map<Boolean, List<Artist>> bandsAndSolo(Stream<Artist> artists) {  
 **return** artists.collect(*partitioningBy*(artist -> artist.isSolo()));  
}

也可以使用方法引用代替Lambda表达式，如下：

**return** artists.collect(*partitioningBy*(Artist::isSolo));

### 5.3.4 数据分组

数据分组是一种更自然的分割数据操作，与将数据分成true和false两部分不同，可以使用任意值对数据分组。比如现在有一个由专辑组成的流，可以按专辑当中的主唱对专辑分组。代码如下：

**public** Map<Artist, List<Album>> albumsByArtist(Stream<Album> albums) {  
 **return** albums.collect(*groupingBy*(album -> album.getMainMusician()));  
}

和其他例子一样，调用流的collect方法，传入一个收集器。groupingBy收集器（如图5-2所示）接受 一个分类函数，用来对数据分组，就像partitioningBy一样，接受一个Predicate对象将数据分成true和false两部分。我们使用的分类器是一个Function对象，和map操作用到的一样。

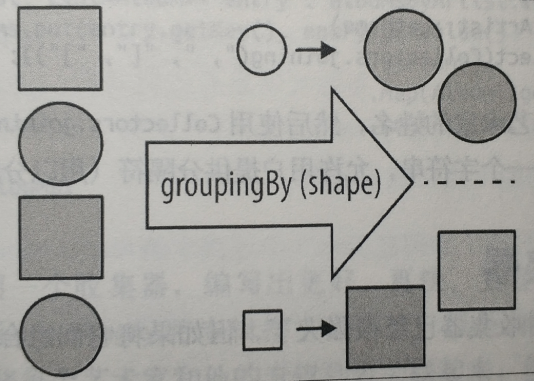


图5-2：groupingBy收集器

读者可能知道SQL中的group by操作，我们的方法是和这类似的一个概念，中不过在Stream类库中实现了而已。

### 5.3.5字符串

很多时候，收集流中的数据都是为了在最后生成一个字符串。假设我们想将参与制作一张专辑的所有艺术家的名字输出为一个格式化好的列表，以专辑Let It Be为例，期望的输出为"[George Harrison, John Lennon, Paul McCartney, Ringo starr, The Beatles]".

在Java 8还未发布前，实现该功能的代码可能如例5-11所示。通过不断迭代列表，使用一个StringBuilder对象来记录结果。每一步都取出一个艺术家的名字，追加到StringBuilder对象。

例5-11 使用for循环格式化艺术家姓名：

StringBuilder builder = **new** StringBuilder(**"["**);  
**for** (Artist artist : artists) {  
 **if** (builder.length() > 1) builder.append(**", "**);  
 builder.append(artist.getName());  
}  
builder.append(**"]"**);  
String result = builder.toString();

显然，这段代码不是非常好。如果不一步步跟踪，很难看出这段代码是干什么的。使用Java8提供的流和收集器就能写出更清晰的代码，如下：

String result = artists.stream().map(Artist::getName).collect(*joining*(**","**, **"["**, **"]"**));

### 5.3.6 组合收集器

combine [kəm'baɪn] vt. 使化合；使联合，使结合 vi. 联合，结合；化合

combiner [kəm'bainə] n. 组合器

虽然读者现在看到的各种收集器已经很强大了，但如果将它们组合起来，会变得更强大。

之前我们使用主唱将专辑分组，现在来考虑如何计算一个艺术家的专辑数量。一个简单的方案是使用前面的方法对专辑先分组后计数，如例5-13所示。

例 5-13 计算每个艺术家专辑数的简单方式：

Map<Artist, List<Album>> albumsByArtist = albums.collect(*groupingBy*(album -> album.getMainMusician()));  
Map<Artist, Integer> numberOfAlbums = **new** HashMap<>();  
**for** (Map.Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {  
 numberOfAlbums.put(entry.getKey(), entry.getValue().size());  
}

这种方式看起来简单，但却有点杂乱无章。这段代码也是命令式的代码，不能自动适应并行化操作。

这里实际上需要另外一个收集器，告诉groupingBy不用为每一个艺术家生成一个专辑列表，只需要对专辑计数就可以了。幸好，核心类库已经提供了一个这样的收集器counting。使用它，可将上述代码重写为下面的样子。

**public** Map<Artist, Long> numberOfAlbums(Stream<Album> albums) {  
 **return** albums.collect(*groupingBy*(album -> album.getMainMusician(), *counting*()));  
}

groupingBy先将元素分成块，每块都与分类函数getMainMusician提供的键值相关联,然后使用下游的另一个收集器收集每块中的元素，最后将结果映射为一个Map.

让我们再看一个例子，这次我们不想生成一组专辑，只希望得到专辑名。这个问题仍然可以用前面的方法解决，先将专辑分组，然后再调整生成的Map中的值，如例5-15所示。

例 5-15 使用简单方式求每个艺术家的专辑名

**public** Map<Artist, List<String>> nameOfAlmbumsDumb(Stream<Album> albums) {  
 Map<Artist, List<Album>> albumsByArtist = albums.collect(*groupingBy*(album -> album.getMainMusician()));  
 Map<Artist, List<String>> nameOfAlbums = **new** HashMap<>();  
 **for** (Map.Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {  
 List<String> names = entry.getValue().stream().map(Album::getName).collect(*toList*());  
 nameOfAlbums.put(entry.getKey(), names);  
 }  
 **return** nameOfAlbums;  
}

同理，我们可以再使用一个收集器，编写出更好、更快、更容易并行处理的代码。我们已经知道，可以使用groupingBy将专辑按主唱分组，但是其输出为一个Map<Artist, List<Album>>对象，它将每个艺术家和他的专辑列表关联起来，但这不是我们想要的，我们想要的是一个包含专辑名的字符串列表。

此时，我们真正想做的是将专辑列表映射为专辑名列表，这里不能直接使用流的map操作，因为列表是由groupingBy生成的。我们需要有一种方法，可以告诉groupingBy将它的值做映射，生成最终结果。

每个收集器都是生成最终值的一剂良方。这里需要两剂配方，一个传给另一个。谢天谢地，Oracle公司的研究员们已经考虑到这种情况，为我们提供了mapping收集器。

mapping允许在收集器的容器上执行类似map的操作。但是需要指明使用什么样的集合类存储结果，比如toList。这些收集器就像乌龟叠罗汉，龟龟相驮以至无穷。

mapping收集器和map方法一样，接受一个Function对象作为参数，经过重构后的代码如下：

**return** albums.collect(*groupingBy*(Album::getMainMusician, *mapping*(Album::getName, *toList*())));

这两个例子中我们都用到了第二个收集器，用以收集最终结果的一个子集。这些收集器叫作下游收集器。收集器是生成最终结果的一剂配方，下游收集器则是生成部分结果的配方，主收集器中会用到下游收集器。这种组合使用收集器的方式，使得它们在Stream类库中的作用更加强大。

那些为基本类型特殊定制的函数，如averaginglnt、sumarizingLong等，事实上和调用特珠Stream上的方法是等价的，加上它们是为了将它们当作下游收集器来使用的。

### 5.3.7 重构和定制收集器

StringCombiner和StringCollector见目录：书中示例用到的JavaBean\

尽管在常用流操作里，Java内置的收集器已经相当好用，但收集器框架本身是极其通用的。JDK提供的收集器没有什么特别的，完全可以定制自己的收集器，而且定制起来相当简单，这就是本节要讲的内容。

StringBuilder builder = **new** StringBuilder(**"["**);  
**for** (Artist artist : artists) {  
 **if** (builder.length() > 1) builder.append(**", "**);  
 builder.append(artist.getName());  
}  
builder.append(**"]"**);  
String result = builder.toString();

上面的例子使用Java 7连接字符串，尽管形式并不优雅。让我们逐步重构这段代码，最终用合适的收集器实现原有代码功能。在工作中没有必要这样做，因为JDK已经提供了一个完美的收集器joining。我们这里只是为了展示如何定制收集器，以及如何使用Java 8提供的新功能来重构遗留代码。

显然，上面的例子可以使用map操作，将包含艺术家的流映射为包含艺术家姓名的流。重构后的代码如下：

StringBuilder builder = **new** StringBuilder(**"["**);  
artists.stream().map(Artist::getName)  
 .forEach(name -> {  
 **if** (builder.length() > 1) builder.append(**", "**);  
 builder.append(name);  
 });  
builder.append(**"]"**);  
String result = builder.toString();

将艺术家映射为姓名，就能更快看出最终是要生成什么，这样代码看起来更清楚一点。可惜forEach方法看起来还是有点笨重，这与我们通过组合高级操作让代码变得易读的目标不符。

暂且不必考虑定制一个收集器，让我们想想怎么通过流上已有的操作来解决该问题。和生成字符串目标最近的操作就是reduce，使用它重构如下：

List<Artist> artists = **null**;  
StringBuilder reduced = artists.stream().map(Artist::getName)  
 .reduce(**new** StringBuilder(),  
 (builder, name) -> {  
 **if** (builder.length() > 0) builder.append(**", "**);  
 builder.append(name);  
 **return** builder;  
 },  
 (left, right) -> left.append(right)  
 );  
  
reduced.insert(0, **"["**);  
reduced.append(**"]"**);  
String result = reduced.toString();

我曾经天真地以为上面的重构会让代码变得更清晰，可惜恰好相反，代码看起来比以前更糟糕。让我们先来看看怎么回事。和前面的例子一样，都调用了stream和map方法，reduce操作生成艺术家姓名列表，艺术家与艺术家之间用“，”分隔。首先创建个StringBuilder对象，该对象是reduce操作的初始状态，然后使用Lambda表达式将姓名连接到builder上。reduce操作的第三个参数也是一个Lambda表达式，接受两个StringBuilder对象做参数，将两者连接起来。最后添加前缀和后缀。

在接下来的重构中，我们还是使用reduce操作，不过需要将杂乱无章的代码隐藏掉——我的意思是使用一个StringCombiner类对细节进行抽象。代码如下：

StringComBinder combined = artists.stream().map(Artist::getName)  
 .reduce(**new** StringCombiner(**","**, **"["**, **"]"**),  
 StringCombiner::add,  
 StringCombiner::merge  
 );  
String result = combined.toString();

尽管代码看起来和上个例子大相径庭，其实背后做的工作是一样的。我们使用reduce操作将姓名和分隔符连接成一个StringBuilder对象。不过这次连接姓名操作被代理到了StringCombiner.add方法，而连接两个连接器操作被StringCombiner.merge方法代理。让我们现在来看看这些方法，先从add方法开始:

add方法返回连接新元素后的结果：

**public** StringCombiner add(String element) {  
 **if** (areAtStart()) {  
 builder.append(prefix); *// 添加前缀* } **else** {  
 builder.append(delim); *// 添加分隔符* }  
 builder.append(element); *// 添加元素* **return this**;  
}

add方法在内部其实将操作代理给一个StringBuilder对象。如果刚开始进行连接，则在最前面添加前缀，否则添加分隔符，然后再添加新的元素。这里返回一个StringCombiner对象，因为这是传给reduce操作所需要的类型。合并代码也是同样的道理，内部将操作代理给StringBuilder对象，merge方法如下：

**public** StringCombiner merge(StringCombiner other) {  
 **builder**.append(other.**builder**);  
 **return this**;  
}

reduce阶段的重构还差一小步就差不多结束了。我们要在最后调用toString方法，将整个步骤串成一个方法链。这很简单，只需要排列好reduce代码，准备好将其转换为Collector API就行了，如下：

使用reduce操作，将工作代理给StringCombiner对象：

String result = artists.stream().map(Artist::getName)  
 .reduce(**new** StringCombiner(**","**, **"["**, **"]"**),  
 StringCombiner::add,  
 StringCombiner::merge  
 )  
 .toString();

现在的代码看起来已经差不多完美了，但是在程序中还是不能重用。因此，我们想将reduce操作重构为一个收集器，在程序中的任何地方都能使用。不妨将这个收集器叫作StringCollector，让我们重构代码使用这个新的收集器，代码如下：

使用定制的收集器StringCollector收集字符串：

String result = artists.stream().map(Artist::getName)  
 .collect(**new** StringCollector(**","**, **"["**, **"]"**));

既然已经将所有对字符串的连接操作代理给了定制的收集器，应用程序就不需要关心StringCollector对象的任何内部细节，它和框架中其他Collector对象用起来是一样的。

先来实现Collector接口(例5-25)，由于Collector接口支持泛型，因此先得确定一些具体的类型：

* 待收集元素的类型，这里是String
* 累加器的类型StringCombiner
* 最终结果的类型，这里依然是String.

例 5-25 定义字符串收集器：

**public class** StringCollector **implements** Collector<String, StringCombiner, String> {  
 @Override  
 **public** Supplier<StringCombiner> supplier() {  
 **return null**;  
 }  
  
 @Override  
 **public** BiConsumer<StringCombiner, String> accumulator() {  
 **return null**;  
 }  
  
 @Override  
 **public** BinaryOperator<StringCombiner> combiner() {  
 **return null**;  
 }  
  
 @Override  
 **public** Function<StringCombiner, String> finisher() {  
 **return null**;  
 }  
  
 @Override  
 **public** Set<Characteristics> characteristics() {  
 **return null**;  
 }  
}

一个收集器由四部分组成。首先是一个Supplier，这是一个工厂方法，用来创建容器，在这个例子中，就是StringCombiner。和reduce操作中的第一个参数类似，它是后续操作的初值,如下：

Supplier是创建容器的工厂：

**public** Supplier<StringCombiner> supplier() {  
 **return** () -> **new** StringCombiner(delim, prefix, suffix);  
}

让我们一边阅读代码，一边看图，这样就能看清到底是怎么工作的。由于收集器可以并行收集，我们要展示的收集操作在两个容器上(比如StringCombiners)并行进行。

收集器的每一个组件都是函数，因此我们使用箭头表示，流中的值用圆圈表示，最终生成的值用椭圆表示。收集操作一开始，Supplier先创建出新的容器(如图5-3)。

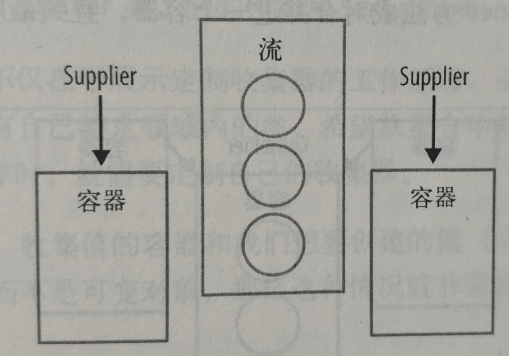


图5-3：Supplier容器

收集器的accumulator的作用和reduce操作的第二个参数一样，它结合之前操作的结果和当前值，生成并返回新的值。这一逻辑已经在StringCombiner的add方法中得以实现，直接引用就好了(如例5-27所示)。

例5-27 accumulator是一个函数，它将当前元素叠加到收集器：

**public** BiConsumer<StringCombiner, String> accumulator() {  
 **return** StringCombiner::add;  
}

这里的accumulator用来将流中的值叠加入容器中(如图5-4所示)。

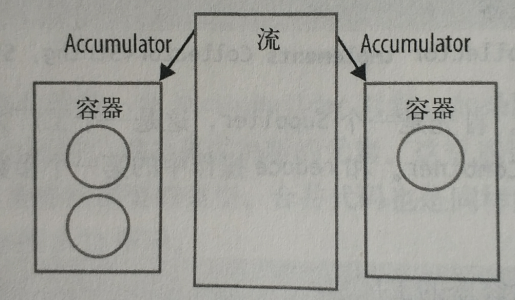
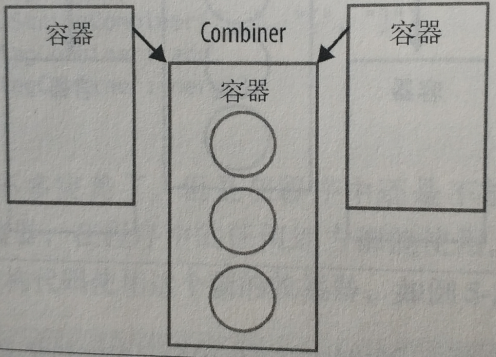


图5-4 Accumulator

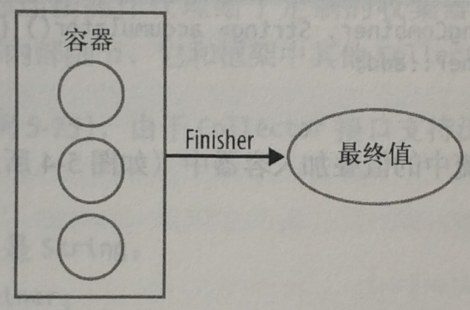
combine方法很像reduce操作的第三个方法。如果有两个容器，我们需要将其合并。同样，在前面的重构中我们已经实现了该功能，直接使用StringCombiner.merge方法就行了，如下：

**public** BinaryOperator<StringCombiner> combiner() {  
 **return** StringCombiner::merge;  
}

在收集阶段，容器被combiner方法成对合并进一个容器，直到最后只剩一个容器为止，如下图：



读者可能 还记得，在使用收集器之前，重构的最后一步将toString方法内联到方法链的末端，这就将StringCombiners转换成了我们想要的字符串，如下：



收集器的finisher方法作用相同。我们已经将流中的值叠加人一个可变容器中，但这还不是我们想要的最终结果。这里调用了finisher方法，以便进行转换。在我们想创建字符串等不可变的值时特别有用，这里容器是可变的。

为了实现finisher方法，只需将该操作代理给已经实现的toString方法即可如下：

finisher方法返回收集操作的最终结果：

**public** Function<StringCombiner, String> finisher() {  
 **return** StringCombiner::toString;  
}

从最后剩下的容器中得到最终结果。

关于收集器，还有一点一直没有提及，那就是**特征**。**特征**是一组描述收集器的对象，框架可以对其适当优化。characteristics方法定义了特征。

在这里我有必要重申，这些代码只作教学用途，和joining收集器的内部实现略有出入。读者也许会认为StringCombiner看起来非常有用，别担心——你没必要亲自去编写，Java8有一个java.util.StringJoiner类，它的作用和StringCombiner一样，有类似的API。

做这些练习的主要目的不仅在于展示定制收集器的工作原理，而且还在于帮助读者编写自己的收集器。特别是你有自己特定领域内的类，希望从集合中构建一个操作，而标准的集合类并没有提供这种操作时，就需要定制自己的收集器。

以StringCombiner为例，收集值的容器和我们想要创建的值(字符串)不一样。如果想要收集的是不可变对象，而不是可变对象，那么这种情况就非常普遍，否则收集操作的每步都需要创建一个新值。

想要收集的最终结果和容器一样是完全有可能的。事实上，如果收集的最终结果是集合，比如toList收集器，就属于这种情况。

此时，finisher方法不需要对容器做任何操作。更正式地说，此时的finisher方法其实是identity函数：它返回传入参数的值。如果这样，收集器就展现出IDENTITY\_FINISH的特征，需要使用characteristics方法声明。

### 5.3.8对收集器的归一化处理

就像之前看到的那样，定制收集器其实不难，但如果你想为自己领域内的类定制一个收集器，不妨考虑一下其他替代方案。最容易想到的方案是构建若干个集合对象，作为参数传给领域内类的构造函数。如果领域内的类包含多种集合，这种方式又简单又适用。

当然，如果领域内的类没有这些集合，需要在已有数据上计算，那这种方法就不合适了。但即使如此，也不见得需要定制二个收集器。你还可以使用reducing收集器，它为流上的归一操作提供了统一实现。下面的示例展示了如何使用reducing收集器编写字符串处理程序。

reducing是一种定制收集器的简便方式：

String result = artists.stream().map(Artist::getName)  
 .collect(Collectors.*reducing*(  
 **new** StringCombiner(**","**, **"["**, **"]"**),  
 name -> **new** StringCombiner(**","**, **"["**, **"]"**).add(name),  
 StringCombiner::merge  
 )  
 )  
 .toString();

这和我在前面例子中讲到的基于reduce操作的实现很像，这点从方法名中就能看出，区别在于Collectors.reducing的第二个参数，我们为流中每个元素创建了唯一的StringCombiner。如果你被这种写法吓到了，或是感到恶心，你不是一个人!这种方式非常低效，这也是我要定制收集器的原因之一。

## 5.4一些细节

Lambda表达式的引入也推动了一些新方法被加入集合类。让我们来看看Map类的一些变化。

构建Map时，为给定值计算键值是常用的操作之一，一个经典的例子就是实现一个缓存。传统的处理方式是先试着从Map中取值，如果没有取到，创建一个新值并返回。

假设使用Map<String, Artist> artistCache定义缓存，我们需要使用费时的数据库操作查询艺术家信息，代码可能如例5-31所示。

例5-31使用显式判断空值的方式缓存

**public** Artist getArtist(String name) {  
 Artist artist = **artistCache**.get(name);  
 **if** (artist == **null**) {  
 artist = readArtistFromDB(name);  
 **artistCache**.put(name, artist);  
 }  
 **return** artist;  
}

compute [kəm'pjuːt] vt. 计算；估算；用计算机计算 vi. 计算；估算；推断 n. 计算；估计；推断

absent [ˈæbsənt] adj. 缺席的；缺少的；心不在焉的；茫然的 vt. 使缺席

Java 8引入了一个新方法computeIfAbsent，该方法接受一个Lambda表达式，值不存在时使用该Lambda表达式计算新值。使用该方法，可将上述代码重写为例5-32所示的形式。

例5-32 使用computeIfAbsent缓存

**public** Artist getArtist(String name) {  
 **return artistCache**.computeIfAbsent(name, **this**::readArtistFromDB);  
}

你可能还希望在值不存在时不计算，为Map接口新增的compute和conputelfAbsent就能处理这些情况。

在工作中，你可能尝试过在Map上迭代。过去的做法是使用value方法返回值的集合，然后在集合上迭代。这样的代码不易读。例5-33展示了本章早些时候介绍的一种方式，创建一个Map，然后统计每个艺术家专辑的数量。

例5-33一种丑陋的迭代Map的方式:

Map<Artist, List<Album>> albumsByArtist = **null**;  
Map<Artist, Integer> countOfAlbums = **new** HashMap<>();  
**for** (Map.Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {  
 Artist artist = entry.getKey();  
 List<Album> albums = entry.getValue();  
 countOfAlbums.put(artist, albums.size());  
}

.

谢天谢地，Java 8为Map接口新增了一个forEach方法，该方法接受一个BiConsumer对象为参数(该对象接受两个参数，返回空)，通过内部迭代编写出易于阅读的代码，关于内部迭代请参考3.1节。使用该方法重写后的代码如例5-34所示。

例5-34使用内部迭代遍历Map里的值

Map<Artist, Integer> countOfAlbums = **new** HashMap<>();  
albumsByArtist.forEach((artist, albums) -> countOfAlbums.put(artist, albums.size()));

## 5.5要点回顾

·方法引用是一种引用方法的轻量级语法，形如：ClassName::methodNane.

·收集器可用来计算流的最终值，是reduce方法的模拟。

·Java 8提供了收集多种容器类型的方式，同时允许用户自定义收集器。

## 5.6练习

1.方法引用

回顾第3章中的例子，使用方法引用改写以下方法

a. 转换大写的map方法。  
 原来的方式如下：  
   
 重写如下：

List<String> uppercaseList = Stream.*of*(**"a"**, **"b"**, **"hello"**)  
 .map(s -> s.toUpperCase())  
 .collect(Collectors.*toList*());

List<String> uppercaseList = Stream.*of*(**"a"**, **"b"**, **"hello"**)  
 .map(String::toUpperCase)  
 .collect(Collectors.*toList*());

b. 使用reduce实现count方法。  
 原来的count方法：  
   
 重写如下：  
   
 这里似乎没有使用到方法引用，第3章中的另一 个count的例子好像也用不 上方法引用：  
 **int** count = Stream.*of*(1, 2, 3).reduce(0, (acc, element) -> acc + element);

long count = allArtists.stream()

.filter(art -> art.isFrom("London"))

.count();

**long** count = allArtists.stream()  
 .reduce(0,  
 (acc, artist) -> {  
 **if** (artist.isFrom(**"London"**)) acc++;  
 **return** acc;  
 },  
 (acc1, acc2) -> acc1 + acc2);

C. 使用flatMap连接列表。

重写如下：

List<Integer> list1 = Arrays.*asList*(1, 2, 3);  
List<Integer> list2 = Arrays.*asList*(4, 5, 6);  
List<Integer> list3 = Stream.*of*(list1, list2).flatMap(list -> list.stream()).collect(Collectors.*toList*());

List<Integer> list1 = Arrays.*asList*(1, 2, 3);  
List<Integer> list2 = Arrays.*asList*(4, 5, 6);  
List<Integer> list3 = Stream.*of*(list1, list2).flatMap(List::stream).collect(Collectors.*toList*());

2.收集器。

a. 找出名字最长的艺术家，分别使用收集器和第3章介绍过的reduce高阶函数实现。然后对比二者的异同：哪一种方式写起来更简单，哪一种方式读起来更简单?以下面的参数为例，该方法的正确返回值为“Stuart Sutcliffe”：

Stream<String> names = Stream.*of* (**"John Lennon"**,**"Paul Mccartney"**,  
 **"George Harrison"**,**"Ringo Starr"**,**"pete Best"**,**"Stuart Sutcliffe"**);  
 答案如下：

import static java.util.Comparator.comparing;

public class LongestName {

private static Comparator<Artist> byNameLength = comparing(artist -> artist.getName().length());

public static Artist byReduce(List<Artist> artists) {

return artists.stream()

.reduce((acc, artist) -> {

return (byNameLength.compare(acc, artist) >= 0) ? acc : artist;

})

.orElseThrow(RuntimeException::new);

}

public static Artist byCollecting(List<Artist> artists) {

return artists.stream()

.collect(Collectors.maxBy(byNameLength))

.orElseThrow(RuntimeException::new);

}

}

b. 假设一个元素为单词的流，计算每个单词出现的次数。假设输入如下，则返回值为一个形如[John ->3, Paul ->2, George -> 1]的Map：

Stream<String> names = Stream. of("John","Paul","George","John","Paul","John");

Stream<String> names = Stream. *of*(**"John"**,**"Paul"**,**"George"**,**"John"**,**"Paul"**,**"John"**);  
Map<String, Long> nameCount = names.collect(*groupingBy*(name -> name, *counting*()));  
StringBuilder sb = **new** StringBuilder(**"["**);  
nameCount.forEach((name, count) -> {  
 **if** (sb.length() > 1) sb.append(**", "**);  
 sb.append(name).append(**" -> "**).append(count);  
});  
sb.append(**"]"**);  
System.***out***.println(sb.toString());

c. 用一个定制的收集器实现Collectors.groupingBy方法，不需要提供一个下游收集器，只需实现一个最简单的即可。别看JDK的源码，这是作弊!提示:可从下面这行代码开始:

public class GroupingBy<T, K> implements Collector<T, Map<K, List<T>, Map<K,List<T>>>

这是一个进阶练习，不妨最后再尝试这道习题。

public class GroupingBy<T, K> implements Collector<T, Map<K, List<T>>, Map<K, List<T>>> {

private final static Set<Characteristics> characteristics = new HashSet<>();

static {

characteristics.add(Characteristics.IDENTITY\_FINISH);

}

private final Function<? super T, ? extends K> classifier;

public GroupingBy(Function<? super T, ? extends K> classifier) {

this.classifier = classifier;

}

@Override

public Supplier<Map<K, List<T>>> supplier() {

return HashMap::new;

}

@Override

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

return (map, element) -> {

K key = classifier.apply(element);

List<T> elements = map.computeIfAbsent(key, k -> new ArrayList<>());

elements.add(element);

};

}

@Override

public BinaryOperator<Map<K, List<T>>> combiner() {

return (left, right) -> {

right.forEach((key, value) -> {

left.merge(key, value, (leftValue, rightValue) -> {

leftValue.addAll(rightValue);

return leftValue;

});

});

return left;

};

}

@Override

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

return map -> map;

}

@Override

public Set<Characteristics> characteristics() {

return characteristics;

}

}

3、改进Map

使用Map的computelfAbsent方法高效计算斐波那契数列。这里的"高效”是指避免将那些较小的序列重复计算多次。

**private static** Map<Integer, List<Integer>> *fibonacciCache* = **new** HashMap<>();  
**public static** List<Integer> getFibonacci(**int** n) {  
 **return** *fibonacciCache*.computeIfAbsent(n, (number) -> *calcFibonacci*(number));  
}  
**public static** List<Integer> calcFibonacci(**int** n) {  
 List<Integer> fibonacciNumbers = **new** ArrayList<>();  
 **int** f1 = 1;  
 **int** f2 = 1;  
 **for** (**int** i = 0; i < n; i++) {  
 **if** (i < 2) {  
 fibonacciNumbers.add(1);  
 } **else** {  
 **int** temp = f1;  
 f1 = f2;  
 f2 = temp + f2;  
 fibonacciNumbers.add(f2);  
 }  
 }  
 **return** fibonacciNumbers;  
}

书中的 答案：

public class Fibonacci {

private final Map<Integer,Long> cache;

public Fibonacci() {

cache = new HashMap<>();

cache.put(0, 0L);

cache.put(1, 1L);

}

public long fibonacci(int x) {

return cache.computeIfAbsent(x, n -> fibonacci(n-1) + fibonacci(n-2));

}

}

# 第6章 数据并行化

前面多次提到, 在Java8中,编写并行化的程序很容易。 这都多亏了第3章介绍的Lambda表达式和流,我们完全不必理会串行或并行,只要告诉程序该做什么就行了。这听起来和长久以来使用Java编程的方式并无区别, 但告诉计算机做什么和怎么做是完全不同的。

从外部迭代到内部迭代的过渡(详见第3章),确实让编写简洁的代码更加容易,但这还不是唯一的好处,另一个好处是程序员不需要手动控制迭代过程了。迭代过程不是非要串行化, 通过改动一个方法调用来告诉计算机我们的意图, 就会出现一个类库指明我们怎么做.

代码的改动微不足道,因此本章主要内容并不在于如何更改代码,而是讲述为什么需要并行化和什么时候会带来性能的提升。要提醒大家的是,本章并不是关于Java 性能的泛泛之谈,我们只关注 Java8轻松提升性能的技术.

## 6.1 并行和并发

快速浏览一下本书的目录结构,读者可能就会发现本章的标题含有并行字样,而第9章的标题则带有并发字样. 别担心,我并不是为了多挣点稿费而将同一个主题写了两次。 并发和并行是两个不同的概念,它们的作用也不一样。

并发是两个任务共享时间段,并行则是两个任务在同一时间发生,比如运行在多核CPU上.如果一个程序要运行两个任务,并且只有一个CPU给它们分配了不同的时间片,那么这就是并发,而不是并行.两者之间的区别如图6-1所示。

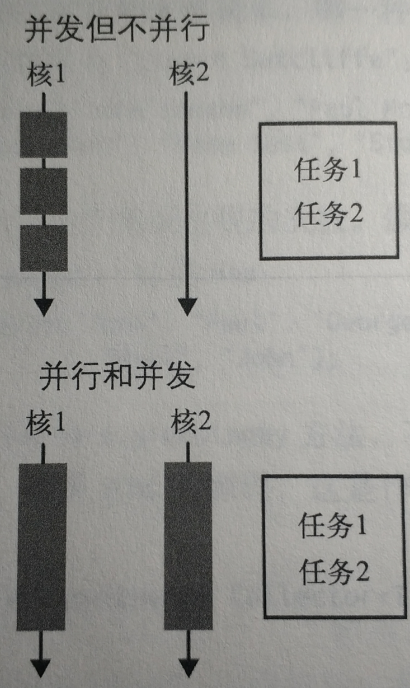


图6-1：并发和并行的区别

并行化是指为缩短任务执行时间,将一个任务分解成几部分, 然后并行执行。 这和顺序执行的任务量是一样的, 区别就像用更多的马来拉车, 花费的时间自然减少了。 实际上, 和顺序执行相比,并行化执行任务时,CPU承载的工作量更大。

本章会讨论一种特殊形式的并行化: 数据并行化。 数据并行化是指将数据分成块, 为每块数据分配单独的处理单元。 还是拿马拉车那个例子打比方, 就像从车里取出一些货物, 放到另一辆车上,两辆马车都沿着同样的路径到达目的地.

当需要在大量数据上执行同样的操作时,数据并行化很管用。它将问题分解为可在多块数据上求解的形式, 然后对每块数据执行运算, 最后将各数据块上得到的结果汇总, 从而获得最终答案.

人们经常拿任务并行化和数据并行化做比较, 在任务并行化中,线程不同,工作各异.我们最常遇到的Java EE应用容器便是任务并行化的例子之一,每个线程不光可以为不同用户服务, 还可以为同一个用户执行不同的任务, 比如登录或往购物车添加商品。

## 6.2 为什么并行化如此重要

过去我们可以指望CPU时钟频率会变得越来越快。1979年, 英特尔公司推出的8086处理器的时钟频率为5 MHz; 到了1993年, 奔腾芯片的速度达到了60 MHz.在21世纪早期,CPU的处理速度一直以这种方式增长.

然而在过去十年中,主流的芯片厂商转向了多核处理器。在写作本书时,服务器通过几个物理单元搭载32或64核的情況已不鲜见,而且,这种趋势尚无减弱的征兆.

这种变化影响到了软件设计。我们不能再依赖提升CPU的时钟频率来提高现有代码的计算能力,需要利用现代CPU的架构,而这唯一的办法就是编写并行化的代码.

大家若已经听过这个消息, 我该是多么欣慰。 事实上, 这一观点在过去几年中,不断地被各种会议的演讲者、技术图书的作者和顾问提及。阿姆达尔定律让我开始关注并行化的重要性。

阿姆达尔定律是一个简单规则,预测了搭载多核处理器的机器提升程序速度的理论最大值.以一段完全串行化的程序为例,如果将其一半改为并行化处理,则不管增加多少处理器, 其理论上的最大速度只是原来的2倍。 有了大量的处理器后,现在这已经是现实了,问题的求解时间将完全取决于它可被分解成几个部分.

以这样的方式思考性能问题, 优化任何和计算相关的任务立即变成了如何有效利用现有硬件的问题. 当然, 不是所有的任务都和计算相关, 本章只关注这类和计算相关的问题.

## 6.3 并行化流操作

parallel ['pærəlel] n. 平行线；对比 vt. 使…与…平行 adj. 平行的；类似的，相同的

并行化操作流只需改变一个方法调用.如果已经有一个Stream对象,调用它的parallel方法就能让其拥有并行操作的能力。如果想从一个集合类创建一个流,调用parallelStream 就能立即获得一个拥有并行能力的流.

让我们先来看一个具体的例子, 例6-1计算了一组专辑的曲目总长度。 它拿到每张专辑的曲目信息,然后得到曲目长度,最后相加得出曲目总长度.

例6-1串行化计算专辑曲目长度

**public int** serialArraySum() {  
 **return albums**.stream()  
 .flatMap(Album::getTracks)  
 .mapToInt(Track::getLength)  
 .sum();  
}

调用parallelStream方法即能并行处理,如例6-2所示,剩余代码都是一样的,并行化就是这么简单!

例6-2并行化计算专辑曲目长度

**public int** parallelArraySum(){  
 **return albums**.parallelStream()  
 .flatMap(Album::getTracks)  
 .mapToInt(Track::getLength)  
 .sum();  
}

读到这里,大家的第一反应可能是立即将手头代码中的stream方法替换为 parallelStream方法,因为这样做简直太简单了! 先别忙, 为了将硬件物尽其用,利用好并行化非常重要,但流类库提供的数据并行化只是其中的一种形式.

我们先要问自己一个问题: 并行化运行基于流的代码是否比串行化运行更快? 这不是一个简单的问题。 回到前面的例子, 哪种方式花的时间更多取决于串行或并行化运行时的环境。

以例6-1和例6-2中的代码为准,在一个四核电脑上,如果有10张专辑,串行化代码的速度是并行化代码速度的8倍;如果将专辑数量增至100张,串行化和并行化速度相当;如果将专辑数量增至10000张,则并行化代码的速度是串行化代码速度的2.5倍.

本章的对比基准只是为了说明问题,如果读者尝试在自己的机器上重现这些实验,得到的结果可能会跟书中的结果大相径庭。

输入流的大小并不是决定并行化是否会带来速度提升的唯一因素,性能还会受到编写代码的方式和核的数量的影响。6.6节会详述和性能有关的细节,但现在还是再来看一个更复杂的例子吧。

## 6.4 模拟系统

并行化流操作的用武之地是使用简单操作处理大量数据,比如模拟系统. 本节我们会搭建一个简易的模拟系统来理解摇骰子,但其中的原理对于大型、真实的系统也适用.

我们这里要讨论的是**蒙特卡洛模拟法**。蒙特卡洛模拟法会重复相同的模拟很多次,每次模拟都使用随机生成的种子。 每次模拟的结果都被记录下来,汇总得到一个对系统的全面模拟.蒙特卡洛模拟法被大量用在工程、金融和科学计算领域.

如果公平地掷两次骰子,然后将赢的一面上的点数相加,就会得到一个2~12的数字.点数的和至少是2,因为骰子六个面上最小的点数是1,而我们将骰子掷了两次;点数的和最大超不过12,因为骰子点数最多的一面也不过6点.我们想要得出点数落在2~12之间每个值的概率.

解决该问题的方法之一是求出掷骰子的所有组合,比如,得到2点的方式是第一次掷得1点,第二次也掷得1点.总共有36种可能的组合,因此,掷得2点的概率就是1/36.

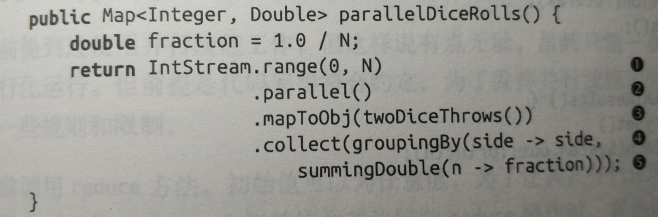
另外ー种解法是使用1到6的随机数模拟掷散子事件,然后用得到毎个点数的次数除以总的投掷次数.这就是一个简单的蒙特卡洛模拟.模拟投掷骰子的次数越多,得到的结果越准确,因此,我们希望尽可能多地增加模拟次数.

range函数就是产生一个0 ~ N的流，包含0不包含N，实验如下：  
IntStream intStream = IntStream.*range*(0, 5);  
**int**[] ints = intStream.toArray();  
System.***out***.println(Arrays.*toString*(ints)); // 输出：[0, 1, 2, 3, 4]

例6-3展示了如何使用流实现蒙特卡洛模拟法。N代表模拟次数,在1处使用IntStream的 range 方法创建大小为N的流,在2处调用 parallel 方法使用流的并行化操作,twoDiceThrows函数模拟了连续掷两次骰子事件,返回值是两次点数之和.在3处使用mapTo0bj方法以便在流上使用该函数.

例6-3 使用蒙特卡洛模拟法并行化模拟掷骰子事件

fraction ['frækʃ(ə)n] n. 分数；部分；小部分；稍微



在4处得到了需要合并的所有结果的流, 使用前一章介绍的groupingBy方法将点数一样的结果合并。 我说过要计算每个点数的出现次数,然后除以总的模拟次数N.在流框架中,将数字映射为1/N并且相加很简单,这和前面说的计算方法是等价的。在5处我们使用summingDouble 方法完成了这一步。 最终的返回值类型是Map<Integer, Double>, 是点数之和到它们的概率的映射.

我得承认这段代码不算儿戏,但使用5行代码即能实现蒙特卡洛模拟法还是很精巧的。重要的是模拟的次数越多, 得到的结果越准确,因此我们运行多次模拟的动机就会更加强烈. 这是一个很好的并行化案列, 并行化能带来速度的提升。

我已经带领读者浏览了整个实现细节, 为了对比,例6-4给出了手动实现并行化蒙特卡洛模拟法的代码。可以看到, 大多数代码都在处理调度和等待线程池中的某项任务完成. 而使用并行化的流时,这些都不用程序员手动管理。

例6-4通过手动使用线程模拟掷骰子事件

public class ManualDiceRolls {

private static final int N = 100000000;

private final double fraction;

private final Map<Integer, Double> results;

private final int numberOfThreads;

private final ExecutorService executor;

private final int workPerThread;

public static void main(String[] args) {

ManualDiceRolls roles = new ManualDiceRolls();

roles.simulateDiceRoles();

}

public ManualDiceRolls() {

fraction = 1.0 / N;

results = new ConcurrentHashMap<>();

numberOfThreads = Runtime.getRuntime().availableProcessors();

executor = Executors.newFixedThreadPool(numberOfThreads);

workPerThread = N / numberOfThreads;

}

public void simulateDiceRoles() {

List<Future<?>> futures = submitJobs();

awaitCompletion(futures);

printResults();

}

private void printResults() {

results.entrySet()

.forEach(System.out::println);

}

private List<Future<?>> submitJobs() {

List<Future<?>> futures = new ArrayList<>();

for (int i = 0; i < numberOfThreads; i++) {

futures.add(executor.submit(makeJob()));

}

return futures;

}

private Runnable makeJob() {

return () -> {

ThreadLocalRandom random = ThreadLocalRandom.current();

for (int i = 0; i < workPerThread; i++) {

int entry = twoDiceThrows(random);

accumulateResult(entry);

}

};

}

private void accumulateResult(int entry) {

results.compute(entry, (key, previous) ->

previous == null ? fraction

: previous + fraction

);

}

private int twoDiceThrows(ThreadLocalRandom random) {

int firstThrow = random.nextInt(1, 7);

int secondThrow = random.nextInt(1, 7);

return firstThrow + secondThrow;

}

private void awaitCompletion(List<Future<?>> futures) {

futures.forEach((future) -> {

try {

future.get();

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

});

executor.shutdown();

}

}

## 6.5 限制

之前提到过使用并行流能工作,但这样说有点无耻。虽然只需一点改动,就能让已有代码并行化运行, 但前提是代码写得符合约定. 为了发挥并行流框架的优势, 写代码时必须遵守一些规则和限制。

之前调用 reduce 方法, 初始值可以为任意值, 为了让其在并行化时能工作正常, 初值必须为组合函数的恒等值。拿恒等值和其他值做reduce操作时,其他值保持不变。比如,使用reduce 操作求和, 组合函数为 (acc, element)-> acc + element,则其初值必须为0, 因为任何数字加0, 值不变。

reduce操作的另一个限制是组合操作必须符合结合律。 这意味着只要序列的值不变, 组合操作的顺序不重要.有点疑惑?别担心!请看例6-5,我们可以改变加法和乘法的顺序,但结果是一样的。

例6-5 加法和乘法满足结合律

(4+ 2) + 1= 4 + (2 + 1) = 7

(4 \* 2) \* 1= 4 \* (2 \* 1)= 8

要避免的是持有锁。 流框架会在需要时,自己处理同步操作,因此程序员没有必要为自己的数据结构加锁。如果你执意为流中要使用的数据结构加锁,比如操作的原始集合,那么有可能是自找麻烦。

在前面我还解释过, 使用parallel方法能轻易将流转换为并行流。如果读者在阅读本书的同时, 还查看了相应的API,那么可能会发现还有一个叫 sequential的方法。 在要对流求值时,不能同时处于两种模式,要么是并行的,要么是串行的.如果同时调用了parallel和sequential方法, 最后调用的那个方法起效。

## 6.6 性能

在前面我简要提及了影响并行流是否比串行流快的一些因素, 现在让我们仔细看看它们。理解哪些能工作,哪些不能工作, 能帮助在如何使用、什么时候使用并行流这一问题上做出明智的决策。影响并行流性能的主要因素有5个,依次分析如下.

* 数据大小

输入数据的大小会影响并行化处理对性能的提升。 将问题分解之后并行化处理,再将结果合并会带来额外的开销. 因此只有数据足够大, 每个数据处理管道花费的时间足够多时, 并行化处理才有意义。6.3节讨论过.

* 源数据结构

每个管道的操作都基于一些初始数据源,通常是集合。将不同的数据源分割相对容易,这里的开销影响了在管道中并行处理数据时到底能带来多少性能上的提升。

* 装箱

处理基本类型比处理装箱类型要快。

* 核的数量

极端情况下, 只有一个核, 因此完全没必要并行化. 显然, 拥有的核越多, 获得潜在性能提升的幅度就越大。在实践中,核的数量不单指你的机器上有多少核,更是指运行时你的机器能使用多少核。 这也就是说同时运行的其他进程, 或者线程关联性(强制线程在某些核或CPU上运行)会影响性能。

* 单元处理开销

比如数据大小,这是一场并行执行花费时间和分解合并操作开销之间的战争。 花在流中每个元素身上的时间越长, 并行操作带来的性能提升越明显。

使用并行流框架，理解如何分解和合并问题是很有帮助的。这让我们能够知悉底层如何工作,但却不必了解框架的细节。

来看一个具体的问题,看看如何分解和合并它。例6-6是并行求和的代码。

例6-6并行求和

**private int** addIntegers(List<Integer> values) {  
 **return** values.parallelStream()  
 .mapToInt(i -> i)  
 .sum();  
}

在底层,并行流还是沿用了fork/join框架。fork递归式地分解问题, 然后每段并行执行，最终由join合并结果,返回最后的值.

图6-2形象地展示了例6-6中代码所示的操作.

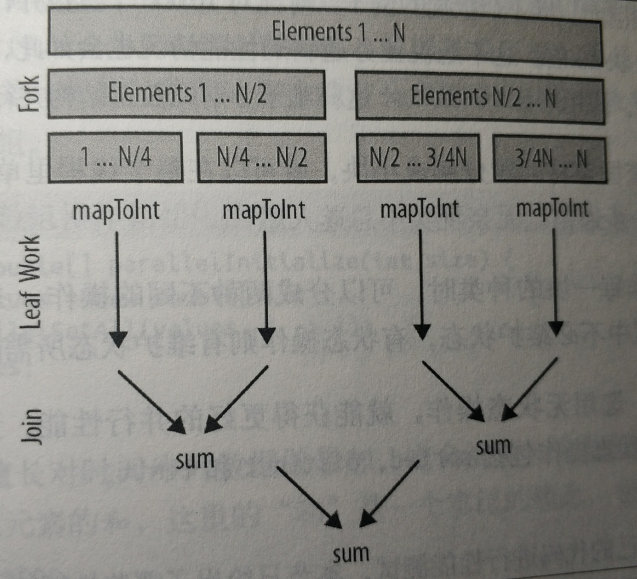


图6-2：使用fork/join分解合并问题

假设并行流将我们的工作分解开,在一个四核的机器上并行执行

1.数据被分成四块.

2.如6-6所示, 计算工作在每个线程里并行执行. 这包括将每个Integer 对象映射为int值,然后在每个线程里将1/4的数字相加。理想情况下,我们希望在这里花的时间越多越好,因为这里是并行操作的最佳场所.

3.然后合并结果。 在例6-6中,就是sum操作,但这也可能是 reduce,collect 或其他终结操作.

根据问题的分解方式, 初始的数据源的特性变得尤其重要,它影响了分解的性能。 直观上看,能重复将数据结构对半分解的难易程度, 决定了分解操作的快慢。能对半分解同时意味着待分解的值能够被等量地分解。

我们可以根据性能的好坏,将核心类库提供的通用数据结构分成以下3组.

* 性能好

ArrayList、数组或IntStream.range，这些数据结构支持随机读取,也就是说它们能轻而易举地被任意分解。

* 性能一般

HashSet, TreeSet，这些数据结构不易公平地被分解,但是大多数时候分解是可能的。

* 性能差

有些数据结构难于分解,比如,可能要花*0*(*N*)的时间复杂度来分解问题.其中包括LinkedList, 对半分解太难了。还有 Streams.iterate 和 BufferedReader.lines, 它们长度未知,因此很难预测该在哪里分解。

初始的数据结构影响巨大。 举一个极端的例子, 对比对10000 个整数并行求和, 使用Arraylist要比使用LinkedList快10 倍。 这不是说业务逻辑的性能情况也会如此, 只是说明了数据结构对于性能的影响之大。使用形如 LinkedList 这样难于分解的数据结构并行运行可能更慢.

理想情况下,一旦流框架将问题分解成小块,就可以在每个线程里单独处理每一小块,线程之间不再需要进一步通信。无奈现实不总遂人愿!

在讨论流中单独操作每一块的种类时,可以分成两种不同的操作：无状态的和有状态的。无状态操作整个过程中不必维护状态,有状态操作则有维护状态所需的开销和限制。

如果能避开有状态,选用无状态操作,就能获得更好的并行性能。无状态操作包括map、filter和flatMap，有状态操作包括 sorted、distinct和limit。

要对自己的代码进行性能测试。本节只给出了哪些性能特征需要调查,但什么都比不上实际的测试和分析。

## 6.7 并行化数组操作

Java8还引入了一些针对数组的并行操作, 脱离流框架也可以使用Lambda表达式。像流框架上的操作一样,这些操作也都是针对数据的并行化操作。让我们看看如何使用这些操作解决那些使用流框架难以解决的问题。

这些操作都在工具类Arrays中,该类还包括Java以前版本中提供的和数组相关的有用方法,表6-1总结了新增的并行化操作.

表6-1:数组上的并行化操作

方法名 操作

parallelPrefix 任意给定一个函数,计算数组的和

parallelSetAll 使用Lambda表达式更新数组元素

parallelSort 并行化对数组元素排序

读者可能以前写过类似例6-7的代码,使用一个for 循环初始化数组. 在这里, 我们用数组下标初始化数组中的每个元素.

imperative[ɪm'perətɪv] adj. 必要的，不可避免的；紧急的；命令的

例6-7 使用for循环初始化数组

public static double[] imperativeInitilize(int size) {

double[] values= new double[size];

for(int i = 0;I < values.length;i++){

values[i]=i;

}

return values;

}

传用 parallelSetALl方法能轻松地并行化该过程, 代码如例6-8所示。 首先提供了一个用于操作的数组,然后传人一个Lambda表达式,根据数组下标计算元素的值.在该例中,数组下标和元素的值是一样的。 使用这些方法有一点要小心: 它们改变了传入的数组, 而没有创建一个新的数组。

例6-8 使用并行化数组操作初始化数组

public static doublel[] parallelInitialize(int size) (

double[] values = new double[size];

Arrays.parallelSetAll(values,i->i);

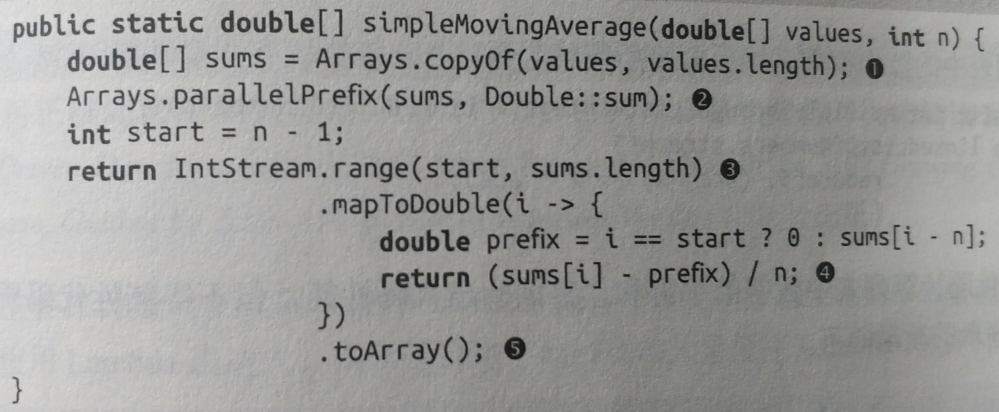
return values;

}

parallelPrefix操作擅长对时间序列数据做累加,它会更新一个数组,将毎一个元素替换为当前元素和其前驱元素的和,这里的“和”是一个宽泛的概念,它不必是加法,可以是任意一个 BinaryOperator.

使用该方法能计算的例子之一是一个简单的滑动平均数.在时间序列上增加一个滑动窗口,计算出窗口中的平均值。如果输人数据为0, 1, 2, 3, 4, 3.5，滑动窗口的大小为3,则简单滑动平均数为1, 2, 3, 3.5，例6-9展示了如何计算滑动平均数。

例6-9 计算简单滑动平均数



这段代码有点复杂,我会分步介绍它是如何工作的。参数n是时间窗口的大小,我们据此计算滑动平均值.由于要使用的并行操作会改变数组内容,为了不修改原有数据,在处复制了一份输人数据。

在处执行并行操作,将数组的元素相加。现在sums变量中保存了求和结果.比如输人0, 1, 2, 3, 4, 3.5, 则计算后的值为0.0, 1.0, 3.0, 6.0, 10.0, 13.5。

现在有了和, 就能计算出时间窗口中的和了, 减去窗口起始位置的元素即可, 除以n即得到平均值。可以使用已有的流中的方法计算该值,那就让我们来试试吧!使用 Intstream.range得到包含所需元素下标的流。

在处使用总和减去窗口起始值, 然后再除以n得到平均值。 最后在处将流转换为数组。

## 6.8要点回顾

* 数据并行化是把工作拆分，同时在多核CPU上执行的方式。
* 如果使用流编写代码，可通过调用parallel或者parallelStream方法实现数据并行化操作。
* 影响性能的五要素是数据大小、源数据结构、值是否装箱、可用的CPU核数量，以及处理每个元素所花的时间。

## 6.9练习

1.例6-10中的代码顺序求流中元素的平方和，将其改为并行处理。

例6-10 顺序求列表中数字的平方和

public static int sequentialSunOfSquares(IntStream range) {

return range.map(x-> x \* x). sum();

}

答：**return** range.parallel().map(x-> x \* x). sum();

2.例6-11中的代码把列表中的数字相乘，然后再将所得结果乘以5。顺序执行这段程序没有问题，但并行执行时有一个缺陷，使用流并行化执行该段代码，并修复缺陷。

例6-11把列表中的数字相乘，然后再将所得结果乘以5，该实现有一个缺陷

public static int multiplyThrough(List<Integer> linkedListOfNumbers) {

return linkedListOfNumbers.stream().reduce(5, (acc, x)-> x \* acc);

}

答：

public static int multiplyThrough(List<Integer> linkedListOfNumbers ) {

return 5 \* linkedListOfNumbers.stream().parallel().reduce(1, (acc, x)-> x \* acc);

}

并行执行时，要确保reduce中的第1个参数跟流中的任意一个元素操作时不改变元素的值，1乘以任何数结果不变，如果是5的话，则不一样。

3.例6-12中的代码计算列表中数字的平方和。尝试改进代码性能，但不得牺牲代码质量。只需要一些简单的改动即可。

例6-12求列表元素的平方和，该实现方式性能不高

public int slowSunOfSquares(List<Integer> linkedListOfNumbers) {

return linkedListOfNumbers.parallelStream()

.map(x -> x \* x)

.reduce(0, (acc, x) -> acc + x);

}

确保将基准代码运行多次，GitHub上提供的示例代码有一份基准数据可供使用。

答案参考中书中的：OptimisationExampleFixed

# 第7章 测试、调试和重构

重构、测试驱动开发(TDD)和持续集成(CI)越来越流行，如果我们需要将Lambda表达式应用于日常编程工作中，就得学会如何为它编写单元测试。

关于如何测试和调试计算机程序的书已经汗牛充栋，本章不打算再一一赘述。如果读者对如何正确地使用测试驱动开发(TDD)感兴趣，我极力推荐大家阅读Kent Beck写的Test-Driven Development, 以及由 Steve Freeman 和 Nat Pryce 写的 Growing Object-Oriented Software, Guided by Tests(两本书均由Addison-Wesley出版社出版)。

本章主要探讨如何在代码中使用Lambda表达式的技术，也会说明什么情况下不应该(直接)使用Lambda表达式。本章还讲述了如何调试大量使用Lambda表达式和流的程序。

先看几个例子，看看如何将现有代码重构为使用Lambda表达式的代码。这部分内容前面已经有所涉及，比如在局部范围内的一些重构，使用流操作替代For循环。本章要讨论的内容更加深入，看看如何使用Lambda表达式提高非集合类代码的质量。

## 7.1重构候选项

使用Lambda表达式重构代码有个时髦的称呼：Lambda化(读作lambda-fi-cation，执行重构的程序员叫作lamb-di-fiers或者有责任心的程序员)。Java 8中的核心类库就曾经历过这样一场重构。在选择内部设计模型时，想想以何种形式向外展示API是大有裨益的。

这里有一些要点，可以帮助读者确定什么时候应该Lambda化自己的应用或类库。其中的每一条都可看作一个局部的反模式或代码异味，借助于Lambda化可以修复。

### 7.1.1进进出出、摇摇晃晃

例7-1是关于如何在程序中记录日志的，我在第4章多次提到这个代码。这段代码先调用isDebugEnabled方法抽取布尔值，用来检査是否启用调试级别，如果启用，则调用Logger对象的相应方法记录日志。如果你发现自己的代码不断地査询和操作某对象，目的只为了在最后给该对象设个值，那么这段代码就本该属于你所操作的对象。

例7-1 Logger对象使用isDebugEnabled属性避免不必要的性能开销

Logger logger = new Logger();

if (logger. isDebugEnabled()) {

logger.debug ("Look at this :"+ expensiveOperation())

}

记录日志本来就是一直以来很难实现的目标，因为地方不同，所需的行为也不一样。本例中，需要根据程序中记录日志的不同位置和要记录的内容生成不同的信息。

这种反模式通过传入代码即数据的方式很容易解决。与其査询并设置一个对象的值，不如传入一个Lambda表达式，该表达式按照计算得出的值执行相应的行为。我将原来的实现代码列在例7-2中，以示提醒。当程序处于调试级别，并且检査是否使用Lambda表达式的逻辑被封装在Logger对象中时，才会调用Lambda表达式。

例7-2使用Lambda表达式简化记录日志代码

Logger logger = new Logger();

Iogger.debug(() -> "Look at this:"+ expensiveoperationo);

上述记录日志的例子也展示了如何使用Lambda表达式更好地面向对象编程(00P)，面向对象编程的核心之一是封装局部状态，比如日志的级别。通常这点做得不是很好，isDebugEnabled方法暴露了内部状态。如果使用Lambda表达式，外面的代码根本不需要检査日志级别。

### 7.1.2孤独的覆盖

这个代码异味是使用继承，其目的只是为了覆盖一个方法。Thread Local就是一个很好的例子。Thread Local能创建一个工厂，为每个线程最多只产生一个值。这是确保非线程安全的类在并发环境下安全使用的一种简单方式。假设要在数据库中査询一个艺术家，但希望每个线程只做一次这种査询，写出的代码可能如例7-3所示。

例7-3 在数据库中查找艺术家

ThreadLocal<Album> thisAlbum = new ThreadLocal<Albun>() {

@Override protected Album initialValue() {

return database. IookupCurrentAlbum();

}

};

在Java 8中，可以为工厂方法withlnitial传入一个Supplier对象的实例来创建对象，如例7-4所示。

例 7-4 使用工厂方法

ThreadLocal<Album> thisAlbum = ThreadLocal.withInitial(() -> database.lookupCurrentAlbum());

我们认为第二个例子优于前一个有以下几个原因。首先，任何已有的Supplier<Album>实例不需要重新封装，就可以在此使用，这鼓励了重用和组合。

在其他都一样的情况下，代码短小精悍就是个优势。更重要的是，这是代码更加清晰的结果，阅读代码时，信噪比降低了。这意味着有更多时间来解决实际问题，而不是把时间花在继承的样板代码上。这样做还有一个优点，JVM会少加载一个类。

对每个试图阅读代码，弄明白代码意图的人来说，也清楚了很多。如果你试着大声念出第二个例子中的单词，能很容易听出是干嘛的，但第一个例子就不行了。

有趣的是，在Java 8以前，这并不是一个反模式，而是惯用的代码编写方式，就像使用匿名内部类传递行为一样，都不是反模式，而是在Java中表达你所想的唯一方式。随着语言的演进，编程习惯也要与时俱进。

### 7.1.3同样的东西写两遍

**不要重复你劳动**(Don't Repeat Yourself, DRY)是一个众所周知的模式，它的反面是同样的东西写两遍(Write Everything Twice, WET)。这种代码异味多见于重复的样板代码，产生了更多需要测试的代码，这样的代码难于重构，一改就坏。

不是所有WET的情况都适合Lambda化。有时，重复是唯一可以避免系统过紧耦合的方式。什么时候该将WET的代码Lambda化?这里有一个信号可以参考。如果有一个整体上大概相似的模式，只是行为上有所不同，就可以试着加入一个Lambda表达式。

让我们看一个更具体的例子。回到我们有关音乐的问题，我想增加一个简单的Order类来计算用户购买专辑的一些有用属性，如计算音乐家人数、曲目和专辑时长等。如果使用命令式Java，编写出的代码如例7-5所示。

例7-5 Order类的命令式实现

public List<Album> albums;

public long countRunningTime() {

long count = 0;

for (Album album : albums)

for (Track track : album. getTrackList())

count += track.getLength();

return count;

}

public long countMusicians() {

long count = 0;

for (Album album : albums)

count += album.getMusicianList().size();

return count;

}

public long countTracks() {

long count = 0;

for (Album album : albums) {

count += album.getTrackList().size();

}

return count;

}

每个方法里，都有样板代码将每个专辑里的属性和总数相加，比如每首曲目的长度或音乐家的人数。我们没有重用共有的概念，写出了更多代码需要测试和维护。可以使用Stream来抽象，使用Java 8中的集合类库来重写上述代码，使之更紧凑。如果直接将上述命令式的代码翻译成使用流的形式，则形如例7-6.

例7-6使用流重构命令式的Order类

public long countRunningTime() {

return albums.stream().mapToLong(album -> album.getTracks().mapToLong(Track::getLength).sum()).sum();

}

public long countMusicians() {

return albums.stream().mapToLong(album -> album.getMusicians().count()).sum();

}

public long countTracks() {

return albums.stream().mapToLong(album -> album.getTracks().count()).sum();

}

然而这段代码仍然有重用可读性的问题，因为有一些抽象和共性只能使用领域内的知识来表达。流不会提供一个方法统计每张专辑上的信息——这是程序员要自己编写的领域知识。这也是在Java 8出现之前很难编写的领域方法，因为每个方法都不一样。

想一下如何实现这样一个函数。我们返回一个long，统计所有专辑的某些特征，还需要一个Lambda表达式，告诉我们统计专辑上的什么信息。也就是说我们的方法需要一个参数，该参数为每张专辑返回一个long，方便的是，Java 8核心类库中已经有了这样个类型ToLongFunction。如图7-1所示，它的类型随参数类型，因此我们要使用的类型为ToLongFunction<Albun>。

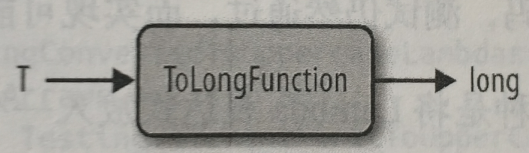


图7-1 ToLongFunction

这些都定下来之后，方法体就自然定下来了。我们将专辑转换成流，将专辑映射为long,然后求和。在实现直接面对客户的代码时，比如countTracks，传入一个代表了领域知识的Lambda表达式，在这里，就是将专辑映射为上面的曲目。例7-7是使用了这种方式转换之后的代码。

例7-7使用领域方法重构Order类

public long countFeature(ToLongFunction<Album> function) {

return albums.stream().mapToLong(function).sum();

}

public long countRunningTime() {

return countFeature(album -> album.getTracks().mapToLong(track -> track.getLength()).sum());

}

public long countMusicians() {

return countFeature(album -> album.getMusicians().count());

}

public long countTracks() {

return countFeature(album -> album.getTracks().count());

}

## 7.2 Lambda表达式的单元测试

单元测试是测试一段代码的行为是否符合预期的方式。

通常，在编写单元测试时，怎么在应用中调用该方法，就怎么在测试中调用。给定一些输入或测试替身，调用这些方法，然后验证结果是否和预期的行为一致。

Lambda表达式给单元测试带来了一些麻烦，Lambda表达式没有名字，无法直接在测试代码中调用。

你可以在测试代码中复制Lambda表达式来测试，但这种方式的副作用是测试的不是真正的实现。假设你修改了实现代码，测试仍然通过，而实现可能早已在做另一件事了。

解决该问题有两种方式。第一种是将Lambda表达式放入一个方法测试，这种方式要测那个方法，而不是Lambda表达式本身。例7-8是一个将一组字符串转换成大写的方法。

例7-8将字符串转换为大写形式

public static List<String> allToUppercase(List<String> words) {

return words.stream().map(string -> string.toUpperCase()).collect(toList());

}

在这段代码中，Lambda表达式唯一的作用就是调用一个Java方法。将该Lambda表达式单独测试是不值得的，它的行为太简单了。

如果换我来测试这段代码，我会将重点放在方法的行为上。比如例7-9测试了流中有多个单词的情况，它们都被转换成对应的大写。

例 7-9 测试大写转换

@Test

public void multipleWordsToUppercase() {

List<String> input = Arrays.asList("a", "b", "hello");

List<String> result = allToUppercase(input);

assertEquals(asList("A", "B", "HELLO"), result);

}

有时候Lambda表达式实现了复杂的功能，它可能包含多个边界情况、使用了多个属性来计算一个非常重要的值。你非常想测试该段代码的行为，但它是一个Lambda表达式，无法引用。

作为例子，让我们来看一个比大写转换更复杂一点的方法。我们要把字符串的第一个字母转换成大写，其他部分保持不变。使用流和Lambda表达式，编写的代码形如例7-10所示。在处使用Lambda表达式做转换。

例7-10 将列表中元素的第一个字母转换成大写

public static List<String> elementFirstToUppercaseLambdas(List<String> words) {

return words.stream().map(word -> {

char firstChar = Character.toUpperCase(word.charAt(0));

return firstChar + word.substring(1);

}).collect(toList());

}

如果要测试这段代码，我们必须创建一个列表，然后将想要测试的各种情况都测试到。例7-11展示了这种方式有多么繁琐，别担心，我们有办法!

例7-11 测试字符串包含两个字符的情况，第一个字母被转换为大写

@Test

public void twoLetterStringConvertedToUppercaseLambdas() {

List<String> input = Arrays.asList("ab");

List<String> result = elementFirstToUppercaseLambdas(input);

assertEquals(asList("Ab"), result);

}

别用Lambda表达式。我知道，在一本介绍如何使用Lambda表达式的书里，这个建议有点奇怪，但是方楔子钉不进圆孔。既然如此，大家一定会问如何测试代码，同时享有Lambda表达式带来的便利？

请用方法引用。任何Lambda表达式都能被改写为普通方法，然后使用方法引用直接引用。

例7-12将Lambda表达式重构为一个方法，然后在主程序中使用，主程序负责转换字符串。

例-12将首字母转换为大写，应用到所有列表元素

**public static** List<String> elementFirstToUppercaseLambdas(List<String> words) {  
 **return** words.stream().map(word -> *firstToUppercase*(word)).collect(*toList*());  
}  
  
**public static** String firstToUppercase(String word) {  
 **char** firstChar = Character.*toUpperCase*(word.charAt(0));  
 **return** firstChar + word.substring(1);  
}

把处理字符串的的逻辑抽取成一个方法后，就可以测试该方法，把所有的边界情况都覆盖到。新的测试用例如例7-13所示。

例 7-13 测试单独的方法

@Test

public void twoLetterStringConvertedToUppercase() {

String input = "ab";

String result = firstToUppercase(input);

assertEquals("Ab", result);

}

## 7.3在测试替身时使用Lambda表达式

编写单元测试的常用方式之一是使用测试替身描述系统中其他模块的期望行为。这种方式很有用，因为单元测试可以脱离其他模块来测试你的类或方法，测试替身让你能用单元测试来实现这种隔离。

测试替身也被称为模拟，事实上测试存根和模拟都属于测试替身。区别 是模拟可以验证代码的行为。读者若想了解 更多有关这方面的信息，请阅读Martin Fowler的相关文章：<http://martinfowler.com/articles/mocksArentStubs.html>

测试代码时，使用Lambda表达式的最简单方式是实现轻量级的测试存根。如果交互的类本身就是一个函数接口、实现这样的存根就非常简单和自然。

在7.1.3节中，讨论过如何将通用的领域逻辑重构为一个countFeature方法，然后使用Lambda表达式实现不同的统计行为。例7-14展示了如何对此编写单元测试。

例7-14使用Lambda表达式编写测试替身，传给countFeature

@Test

public void canCountFeatures() {

OrderDomain order = new OrderDomain(asList(

new Album("Exile on Main St."),

new Album("Beggars Banquet"),

new Album("Afternath").

new Album("Let it Bleed")));

assertEquals(8, order.countFeature(album -> 2));

}

对于countFeature方法的期望行为是为传入的专辑返回某个数值。这里传入4张专辑，测试存根中为每张专辑返回2，然后断言该方法返回8，即2×4。如果要向代码传入一个Lambda表达式，最好确保Lambda表达式也通过测试。

多数的测试替身都很复杂，使用Mockito这样的框架有助于更容易地产生测试替身。让我考虑一种简单情形，为List生成测试替身。我们不想返回List本上的长度，而是返回另一个List的长度，为了模拟List的size方法，我们不想只给出答案，还想做一些操作，因此传入一个Lambda表达式，如例7-15所示。

例7-15 结合Mockito框架使用Lambda表达式

List<String> list = mock(List.class);

when(list.size())thenAnswer(inv -> otherList.size());

assertEquals(3, list.size());

Mockito使用Answer接口允许用户提供其他行为，换句话说，这是我们的老朋友：代码即数据。之所以在这里能使用Lambda表达式，是因为Answer本身就是一个函数接口。

## 7.4惰性求值和调试

调试时通常会设置断点，单步跟踪程序的每一步。使用流时，调试可能会变得更加复杂，因为迭代已交由类库控制，而且很多流操作是惰性求值的。

在传统的命令式编程看来，代码就是达到某种目的的一系列行动，在行动前后査看程序状态是有意义的。在Java 8中，你仍然可以使用IDE提供的各种调试工具，但有时需要调整实现方式，以期达到更好的结果。

## 7.5日志和打印消息

假设你要在集合上进行大量操作，你要调试代码，你希望看到每一步操作的结果是什么。可以在每一步打印出集合中的值，这在流中很难做到，因为一些中间步骤是惰性求值的。

让我们通过第3章介绍的命令式版本的国际报告程序，看看如何记录中间值。考虑到读者可能已经忘记这个程序，我们再来解释一下这个程序的意图，该程序找出了专辑上每位艺术家来自哪个国家。在例7-16中，我们将找到的国家信息记录到日志中。

例7-16记录中间值，以便调试for循环

Set<String> nationalities = new HashSet<>();

for (Artist artist : album.getMusicianList()) {

if (artist.getName().startsWith("The")) {

String nationality = artist.getNationality();

System.out.println("Found nationality" + nationality);

nationalities.add(nationality);

}

}

return nationalities;

现在可以使用forEach方法打印出流中的值，这同时会触发求值过程。但是这样的操作有个缺点我们无法再继续操作流了，流只能使用一次。如果我们还想继续，必须重新创建流。例7-17展示了这样的代码会有多难看。

例7-17使用forEach记录中间值，这种方式有点幼稚

album.getMusicians().filter(artist -> artist.getName().startsWith("The"))

.map(artist -> artist.getNationality())

.forEach(nationality -> System.out.println("Found: " + nationality));

Set<String> nationalities

= album.getMusicians().filter(artist -> artist.getName().startsWith("The"))

.map(artist -> artist.getNationality())

.collect(Collectors.toSet());

## 7.6 解决方案：peek

幸运的是，流有一个方法让你能査看每个值，同时能继续操作流。这就是peek方法。例7-18使用peek方法重写了前面的例子，输出流中的值，同时避免了重复的流操作。

例7-18 使用peek方法记录中间值

Set<String> nationalities

= album.getMusicians().filter(artist -> artist.getName().startsWith("The"))

.map(artist -> artist.getNationality())

.peek(nation -> System.out.println("Found: " + nation))

.collect(Collectors.toSet());

使用peek方法还能以同样的方式，将输出定向到现有的日志系统中，比如log4j、java.util.logging或者slf4j。

## 7.7在流中间设置断点

记录日志这是peek方法的用途之一。为了像调试循环那样一步一步跟踪，可在peek方法中加入断点，这样就能逐个调试流中的元素了。

此时，peek方法可知包含一个空的方法体，只要能设置断点就行。有一些调试器不允许在空的方法体中设置断点，此时，我将值简单地映射为其本身，这样就有地方设置断点了，虽然这样做不够完美，但只要能工作就行。

## 7.8要点回顾

* 重构遗留代码时考虑如何使用Lambda表达式，有一些通用的模式。
* 如果想要对复杂一点的Lambda表达式编写单元测试，将其抽取成一个常规的方法。
* peek方法能记录中间值，在调试时非常有用。

# 第8章 设计和架构的原则

软件开发最重要的设计工具不是什么技术，而是一颗在设计原则方面训练有素的头脑。——Craig Larman

通过前面的学习，我们认识到Lambda表达式是对Java语言的一点简单改进，在JDK标准类库中，运行它的方式各种各样。但是大多数Java代码都不是由开发JDK的程序员写的，而是像你我这样的普通程序员。为了最大限度发挥Lambda表达式的优势，大家需要将其引人已有代码中。作为一名职业Java程序员，Lambda表达式没有什么特别的，和接口、类一样，它只是程序员工具箱中的一件新工具。

本章将探索如何使用Lambda表达式实现SOLID原则，该原则是开发良好面向对象程序的准则。使用Lambda表达式，还能改进一些现有的设计模式，本章也会为大家简要介绍几个这样的例子。

和同事一起工作时，肯定会遇到这样的情况：你实现了一个新功能或修复了一个缺陷，并且对自己的修改很满意。但其他人看了你的代码后——也许发生在代码审查环节，完全不买账!对于什么是好代码，什么是坏代码，存在分歧很正常。

大多数时候，人们意见不统一，是他们各自都有自己的想法。审查你代码的人可能会选择另一种实现方式，这并不是说你们俩谁对谁错。引入Lambda表达式后，又多了一个话题。这并不是说该功能本身有多复杂，或者需要花大力气去争论，而是人们在讨论设计问题时又多了一项谈资。

本章旨在帮助大家写出优秀的程序，我会给出一些良好的设计原则和模式，在此基础之上，就能开发出可维护且十分可靠的程序。我们不光会用到JDK提供的崭新类库，而且会教大家如何在自己的领域和应用程序中使用Lambda表达式。

## 8.1 Lambda表达式改变了设计模式

设计模式是人们熟悉的另一种设计思想，它是软件架构中解决通用问题的模板。如果碰到一个问题，并且恰好熟悉一个与之适应的模式，就能直接应用该模式来解决问题。从某种程度上来说，设计模式将解决特定问题的最佳实践途径固定了下来。

当然，没有永远的最佳实践。以曾经风靡一时的单例模式为例，该模式确保只产生一个对象实例。在过去十年中，人们批评它让程序变得更脆弱，且难于测试。敏捷开发的流行，让测试显得更加重要，单例模式的这个问题把它变成了一个**反模式**：一种应该避免使用的模式。

本书的重点并不是讨论设计模式如何变得过时，相反，我们讨论的是如何使用Lambda表达式，让现有设计模式变得更好、更简单，或者在某些情况下，有了不同的实现方式。Java 8引入的新语言特性是所有这些设计模式变化的推动因素。

### 8.1.1命令者模式

**命令者**是一个对象，它封装了调用另一个方法的所有细节，**命令者模式**使用该对象，可以编写出根据运行期条件，顺序调用方法的一般化代码，命令者模式中有四个类参与其中，如图8-1所示。

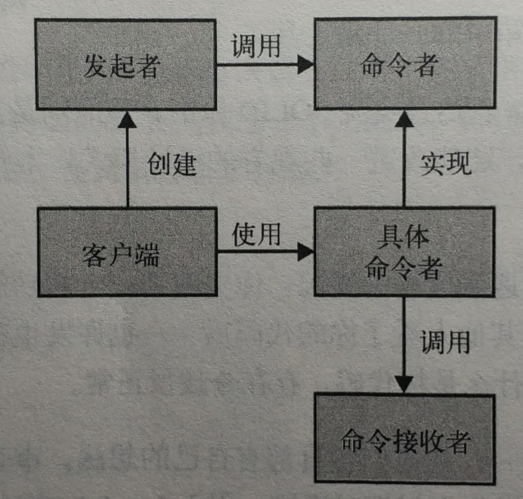


图8-1：命令者模式

* 命令接收者

执行实际任务。

* 命令者

封装了所有调用命令执行者的信息

* 发起者

控制一个或多个命令的顺序和执行

* 客户端

创建具体的命令者实例

看一个命令者模式的具体例子，看看如何使用Lambda表达式改进该模式。假设有一个GUI Editor组件，在上面可以执行open、save等一系列操作，如例8-1所示。现在我们想实现宏功能——也就是说，可以将一系列操作录制下来，日后作为一个操作执行，这就是我们的命令接收者。

例8-1 文本编辑器可能 具有的一 般功能

public interface Editor {

public void save();

public void open();

public void close();

}

在该例子中，像open、save这样的操作称为命令，我们需要一个统一的接口来概括这些不同的操作，我将这个接口叫作Action，它代表了一个操作。所有的命令都要实现该接口（例8-2）。

例8-2 所有操作均实现Action接口

public interface Action {

public void perform();

}

现在让每个操作都实现该接口，这些类要做的只是在Action接口中调用Editor类中的一个方法。我将遵循恰当的命名规范，用类名代表操作，比如save方法对应Save类。例8-3和例8-4是定义好的命令对象。

例8-3保存操作代理给Editor方法

public class Save implements Action {

private final Editor editor;

public Save(Editor editor) {

this.editor = editor;

}

@Override

public void perform() {

editor.save();

}

}

例8-4 打开文件操作代理给Editor方法

public class Open implements Action {

private final Editor editor;

public Open(Editor editor) {

this.editor = editor;

}

@Override

public void perform() {

editor.open();

}

}

现在可以实现Macro类了，该类record操作，然后一起运行。我们使用List保存操作序列，然后调用forEach方法按顺序执行每一个Action，例8-5就是我们的命令发起者。

例8-5 包含操作序列的宏，可按顺序执行操作

public class Macro {

private final List<Action> actions;

public Macro() {

actions = new ArrayList<>();

}

public void record(Action action) {

actions.add(action);

}

public void run() {

actions.forEach(Action::perform);

}

}

在构建宏时，将每一个命令实例加入Macro对象的列表，然后运行宏，就会按顺序执行每一条命令。我是个“懒惰的”程序员，喜欢将通用的工作流定义成宏。我说“懒惰”了吗?我的意思其实是提高工作效率。例8-6展示了如何在用户代码中使用Macro对象。

例8-6 使用命令者模式构建宏

Macro macro = new Macro();

macro.record(new Open(editor));

macro.record(new Save(editor));

macro.record(new Close(editor));

macro.run();

Lambda表达式能做点什么呢?事实上，所有的命令类，Save、Open都是Lambda表达式，只是暂时藏在类的外壳下。它们是一些行为，我们通过创建类将它们在对象之间传递。Lambda表达式能让这个模式变得非常简单，我们可以扔掉这些类。例8-7展示了去掉命令类，使用Lambda表达式后的程序。

例8-7使用Lambda表达式构建宏

Macro macro = new Macro();

macro. record(() -> editor. open());

macro. record(()-> editor. save());

macro. record(() -> editor. close());

macro. run();

事实上，如果意识到这些Lambda表达式的作用只是调用了一个方法，还能让问题变得更简单。我们可以使用方法引用将命令和宏对象关联起来(如例8-8所示)。

例8-8使用方法引用构建宏

Macro macro = new Macro();

macro. record(() -> editor::open);

macro. record(()-> editor::save);

macro. record(() -> editor::close);

macro. run();

命令者模式只是一个可怜的程序员使用Lambda表达式的起点。使用Lambda表达式或是方法引用，能让代码更简洁，去除了大量样板代码，让代码意图更加明显。

宏只是使用命令者模式的一个例子，它被大量用在实现组件化的图形界面系统、撤销功能、线程池、事务和向导中。

在核心Java中，已经有一个和Action接口结构一致的函数接口——Runnable。我们可以在实现上述宏程序中直接使用该接口，但在这个例子中，似乎Action是一个更符合我们待解问题的词汇，因此我们创建了自己的接口。

### 8.1.2策略模式

策略模式能在运行时改变软件的算法行为。如何实现策略模式根据你的情况而定，但其主要思想是定义一个通用的问题，使用不同的算法来实现，然后将这些算法都封装在一个统一接口的背后。

文件压缩就是一个很好的例子，我们提供给用户各种压缩文件的方式，可以使用zip算法，也可以使用gzip算法，我们实现一个通用的Compressor类，能以任何一种算法压缩文件。

首先，为我们的策略定义API(参见图8-2)，我把它叫作CompressionStrategy，每一种文件压缩算法都要实现该接口。该接口有一个compress方法，接受并返回一个OutputStream对象，返回的就是压缩后的OutputStream (如例8-9所示)。

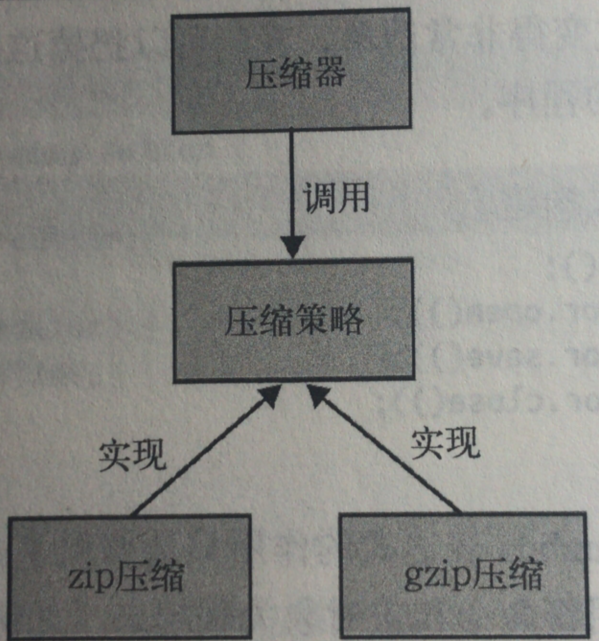


图8-2策略模式

例8-9定义压缩数据的策略接口

public interface CompressionStrategy {

public OutputStream compress(OutputStream data) throws IOException;

}

我们有两个类实现了该接口，分别代表gzip和ZIP算法，使用Java内置的类实现gzip(例8-10)和ZIP(例8-11)算法。

例8-10 使用gzip算法压缩数据

public class GzipCompressionStrategy implements CompressionStrategy {

@Override

public OutputStream compress(OutputStream data) throws IOException {

return new GZIPOutputStream(data);

}

}

例8-11使用zip算法压缩数据

public class ZipCompressionStrategy implements CompressionStrategy {

@Override

public OutputStream compress(OutputStream data) throws IOException {

return new ZipOutputStream(data);

}

}

现在可以动手实现Compressor类了，这里就是使用策略模式的地方。该类有一个compress方法，读入文件，压缩后输出。它的构造函数有一个CompressionStrategy参数，调用代码可以在运行期使用该参数决定使用哪种压缩策略，比如，可以等待用户输入选择(如例8-12所示)。

例8-12在构造类时提供压缩策略

public class Compressor {

private final CompressionStrategy strategy;

public Compressor(CompressionStrategy strategy) {

this.strategy = strategy;

}

public void compress(Path inFile, File outFile) throws IOException {

try (OutputStream outStream = new FileOutputStream(outFile)) {

Files.copy(inFile, strategy.compress(outStream));

}

}

}

如果使用这种传统的策略模式实现方式，可以编写客户代码创建一个新的Compressor，并且使用任何我们想要的策略(如例8-13所示)。

例8-13使用具体的策略类初始化Compressor

Compressor gzipCompressor = new Compressor(new GzipCompressionStrategy());

gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(new ZipCompressionStrategy());

zipCompressor.compress(inFile, outFile);

和前面讨论的命令者模式一样，使用Lambda表达式或者方法引用可以去掉样板代码。在这里，我们可以去掉具体的策略实现，使用一个方法实现算法，这里的算法由构造函数中对应的OutputStream实现。使用这种方式，可以完全舍弃GzipCompresstionStrategy和ZipCompressionStrategy类。例8-14展示了使用方法引用后的代码。

8-14使用方法引用初始化Compressor

Compressor gzipCompressor = new Compressor(GZIPOutputStream::new);

gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(ZipOutputStream::new);

zipCompressor.compress(inFile, outFile);

### 8.1.3观察者模式

观察者模式是另一种可被Lambda表达式简化和改进的行为模式。在观察者模式中，被观察者持有一个观察者列表，当被观察者的状态发生改变，会通知观察者。观察者模式被大量应用于基于MVC的GUI工具中，以此让模型状态发生变化时，自动刷新视图模块，达到二者之间的解耦。

NASA美国国家航空航天局（英语：National Aeronautics and Space Administration，简称NASA /ˈnæsə/），又称美国宇航局、美国太空总署

观看GUI模块自动刷新有点枯燥，我们要观察的对象是月球! NASA和外星人都对登陆到月球上的东西感兴趣，都希望可以记录这些信息。NASA希望确保阿波罗号上的航天局成功登月!外星人则希望在NASA注意力分散之时进犯地球。

让我们先来定义观察者的API,这里我将观察者称作LandingObserver.它只有一 个observeLanding方法，当有东西登陆到月球上时会调用该方法(例8-15)。

例8-15用于观察登陆到月球的组织的接口

public interface LandingObserver {

public void observeLanding(String name);

}

被观察者是月球Moon，它持有一组LandingObserver实例，有东西着陆时会通知这些观察者，还可以增加新的LandingObserver实例观测Moon对象(例8-16)。

例8-16 Moon类——当然不如现实世界中那么完美

public class Moon {

private final List<LandingObserver> observers = new ArrayList<>();

public void land(String name) {

for (LandingObserver observer : observers) {

observer.observeLanding(name);

}

}

public void startSpying(LandingObserver observer) { // Spying n. 间谍活动；侦探 v. 从事间谍活动；发现；仔细察看；暗中监视（spy的ing形式）

observers.add(observer);

}

}

我们有两个具体的类实现了LandingObserver接口，分别代表外星人(例8-17)和NASA(例8-18)检测着陆情况。前面提到过，监测到登陆后它们有不同的反应。

例8-17外星人观察到人类登陆月球

public class Aliens implements LandingObserver { // alien ['eɪlɪən]n. 外国人，外侨；外星人

@Override

public void observeLanding(String name) {

if (name.contains("Apollo")) { // Apollo[ə'pɒləʊ]n. 阿波罗（美国阿波罗登月计划）

System.out.println("They're distracted, lets invade earth!"); // 他们分心了，让我们入侵地球吧!

}

}

}

例 8-18 NASA也能观察到有人登陆月球

public class Nasa implements LandingObserver {

@Override

public void observeLanding(String name) {

if (name.contains("Apollo")) {

System.out.println("We made it!"); // 我们实现了！

}

}

}

和前面 的模式类似，在传统的例子中，用户代码需要有一层模板类，如果使用Lambda表达式，就不用编写这些类了（如例8-19和例8-20所示）。

例8-19 使用类的方式构建用户代码

Moon moon = new Moon();

moon.startSpying(new Nasa());

moon.startSpying(new Aliens());

moon.land("An asteroid"); // asteroid['æstərɒɪd]n. [天] 小行星

moon.land("Apollo 11");

例 8-20 使用Lambda表达式构建用户代码

Moon moon = new Moon();

moon.startSpying(name -> {

if (name.contains("Apollo"))

System.out.println("We made it!");

});

moon.startSpying(name -> {

if (name.contains("Apollo"))

System.out.println("They're distracted, lets invade earth!");

});

moon.land("An asteroid");

moon.land("Apollo 11");

还有一点值得思考，无论使用观察者模式或策略模式，实现时采用Lambda表达式还是传统的类，取决于策略和观察者代码的复杂度。我这里所举的例子代码很简单，只是一两个方法调用，很适合展示新的语言特性。然而在有些情况下，观察者本身就是一个很复杂的类，这时将很多代码塞进一个方法中会大大降低代码的可读性。

从某种角度来说，**将大量代码塞进一个方法会让可读性变差**是决定如何使用Lambda表达式的黄金法则。之所以不在这里过分强调，是因为这也是编写一般方法时的黄金法则!

### 8.1.4模板方法模式

开发软件时一个常见的情况是有一个通用的算法，只是步骤上略有不同。我们希望不同的实现能够遵守通用模式，保证它们使用了同一个算法，也是为了让代码更加易读。一旦你从整体上理解了算法，就能更容易理解其各种实现。

模板方法模式是为这些情况设计的：整体算法的设计是一个抽象类，它有一系列抽象方法，代表算法中可被定制的步骤，同时这个类中包含了一些通用代码。算法的每一个变种由具体的类实现，它们重写了抽象方法，提供了相应的实现。

让我们假想一个情境来搞明白这是怎么回事。假设我们是一家银行，需要对公众、公司和职员放贷。放贷程序大体一致——验明身份、信用记录和收入记录。这些信息来源不一，衡量标准也不一样。你可以査看一个家庭的账单来核对个人身份；公司都在官方机构注册过，比如美国的SEC、英国的Companies House。

loan [ləʊn] n. 贷款；借款 vi. 借出 vt. 借；借给

application [ˌæplɪ'keɪʃ(ə)n] n. 应用；申请；应用程序

我们先使用一个抽象类LoanApplication来控制算法结构，该类包含一些贷款调査结果报告的通用代码。根据不同的申请人，有不同的类：CompanyLoanApplication、PersonalLoanApplication 和 EmployeeLoanApplication。 例 8-21 展示了LoanApplication 类的结构。

例8-21使用模板方法模式描述申请贷款过程

public abstract class LoanApplication {

public void checkLoanApplication() throws ApplicationDenied {

checkIdentity(); // 检查身份

checkCreditHistory(); // 检查信用历史

checkIncomeHistory(); // 检查收入历史

reportFindings(); // 报告调查的结果

}

protected abstract void checkIdentity() throws ApplicationDenied;

protected abstract void checkIncomeHistory() throws ApplicationDenied;

protected abstract void checkCreditHistory() throws ApplicationDenied;

private void reportFindings() { }

}

CompanyLoanApplicatton的checkldentity方法在Companies House等注册公司数据库中査找相关信息。checkIncomeHistory方法评估公司的现有利润、损益表和资产负债表。checkCreditHistory方法则査看现有的坏账和未偿债务。

PersonalLoanApplication的checkIdentity方法通过分析客户提供的纸本结算单，确认客户地址是否真实有效，checkIncomeHistory方法通过检査工资条判断客户是否仍被雇佣。checkCreditHistory方法则会将工作交给外部的信用卡支付提供商。

EmployeeLoanApplication就是没有査阅员工历史功能的PersonalLoanApplication。为了方便起见，我们的银行在雇佣员工时会查阅所有员工的收入记录（例8-22）。

例8-22员工申请贷款是个人申请的一种特殊情况

public class EmployeeLoanApplication extends PersonalLoanApplication {

@Override

protected void checkIncomeHistory() {

// They work for us! （这是自己人）

}

}

使用Lambda表达式和方法引用，我们能换个角度思考模板方法模式，实现方式也跟以前不一样。模板方法模式真正要做的是将一组方法调用按一定顺序组织起来。如果用函数接口表示函数，用Lambda表达式或者方法引用实现这些接口，相比使用继承构建算法，就会得到极大的灵活性。让我们看看如何使用这种方式实现LoanApplication算法，请看例8-23！

例8-23员工申请贷款的例子

public class LoanApplication {

private final Criteria identity; // criteria [kraɪ'tɪərɪə] n. 标准，条件（criterion的复数）

private final Criteria creditHistory;

private final Criteria incomeHistory;

public LoanApplication(Criteria identity, Criteria creditHistory, Criteria incomeHistory) {

this.identity = identity;

this.creditHistory = creditHistory;

this.incomeHistory = incomeHistory;

}

public void checkLoanApplication() throws ApplicationDenied {

identity.check();

creditHistory.check();

incomeHistory.check();

reportFindings();

}

private void reportFindings() { }

}

正如读者所见，这里没有使用一系列的抽象方法，而是多出一些属性identity、creditHistory和incomeHistory。每一个属性都实现了函数接口Criteria，该接口检査一项标准，如果不达标就抛出一个问题域里的异常。我们也可以选择从check方法返回一个类来表示成功或失败，但是沿用异常更加符合先前的实现(如例8-24所示)。

例8-24如果申请失败，函数接口Criteria抛出异常

public interface Criteria {

public void check() throws ApplicationDenied;

}

采用这种方式，而不是基于继承的模式的好处是不需要在LoanApplication及其子类中实现算法，分配功能时有了更大的灵活性。比如，我们想让Company类负责所有的检查，那么Company类就会多出一系列方法，如例8-25所示。

例8-25 Company类中的检査方法

public class Company {

public void checkIdentity() throws ApplicationDenied { }

public void checkProfitAndLoss() throws ApplicationDenied { }

public void checkHistoricalDebt() throws ApplicationDenied { }

}

现在只需为CompanyLoanApplication类传入对应的方法引用，如例8-26所示。

例8-26 CompanyLoanApplication类声明了对应的检査方法

public class CompanyLoanApplication extends LoanApplication {

public CompanyLoanApplication(Company company) {

super(company::checkIdentity,

company::checkHistoricalDebt,

company::checkProfitAndLoss);

}

}

将行为分配给Company类的原因是各个国家之间确认公司信息的方式不同。在英国，Companies House规范了注册公司信息的地址，但在美国，各个州的政策是不一样的。

使用函数接口实现检査方法并没有排除继承的方式。我们可以显式地在这些类中使用Lambda表达式或者方法引用。

我们也不需要强制EmployeeLoanApplication继承PersonalLoanApplication来达到复用，可以对同一个方法传递引用。它们之间是否天然存在继承关系取决于员工的借贷是否是普通人借贷这种特殊情况，或者是另外一种不同类型的借贷。因此，使用这种方式能让我们更加紧密地为问题建模。

## 8.2使用Lambda表达式的领域专用语言

**领域专用语言(DSL)**是针对软件系统中某特定部分的编程语言。它们通常比较小巧，表达能力也不如Java这样能应对大多数编程任务的通用语言强。DSL高度专用：不求面面俱到，但求有所专长。

人们通常将DSL分为两类：内部DSL和外部DSL。外部DSL脱离程序源码编写，然后单独解析和实现。比如级联样式表(CSS)和正则表达式，就是常用的外部DSL。