# 为什么要关心java8

## 1.流处理

流是一系列数据项，一次只生成一项。程序可以从输入流中一个一个读取数据项，然后以同样的方式将数据项写入输出流。一个程序的输出流很可能是另一个程序的输入流。Java 8在java.util.stream中添加了一个Stream API。

新的Stream API和Java现有的集合API的行为差不多：它们都能够访问数据项目的序列。不过，现在最好记得， Collection主要是为了存储和访问数据，而Stream则主要用于描述对数据的计算。这里的关键点在于， Stream允许并提倡并行处理一个Stream中的元素。虽然可能乍看上去有点儿怪，但筛选一个Collection（将上一节的filterApples应用在一个List上）的最快方法常常是将其转换为Stream，进行并行处理，然后再转换回List

|  |
| --- |
| 顺序处理：  List<Apple> heavyApples =inventory.**stream()**.filter((Apple a) -> a.getWeight() > 150) .collect(toList());  并行处理：  List<Apple> heavyApples =inventory.**parallelStream()**.filter((Apple a) -> a.getWeight() > 150).collect(toList()); |

## 2.行为参数化

Java 8中增加的另一个编程概念是通过API来传递代码的能力。java 8增加了把方法（你的代码）作为参数传递给另一个方法的能力。这一概念称为行为参数化。Stream API就是构建在通过传递代码使操作行为实现参数化的思想上的。

## 3.并行与共享的可变数据

Java 8的流实现并行比Java现有的线程API更容易，因此，尽管可以使用synchronized来打破“不能有共享的可变数据”这一规则，但这相当于是在和整个体系作对，因为它使所有围绕这一规则做出的优化都失去意义了。在多个处理器内核之间使用synchronized，其代价往往比你预期的要大得多，因为同步迫使代码按照顺序执行，而这与并行处理的宗旨相悖。  
这两个要点（没有共享的可变数据，将方法和函数即代码传递给其他方法的能力）是我们平  
常所说的函数式编程范式的基石。

## 4.方法和Lambda最为一等公民

4.1 什么是一等公民？

想想Java程序可能操作的值吧。首先有原始值，比如42（int类型）和3.14（double类型）。其次，值可以是对象（更严格地说是对象的引用）。

编程语言的整个目的就在于操作值，要是按照历史上编程语言的传统，这些值因此被称为一等值（或一等公民，这个术语是从20世纪60年代美国民权运动中借用来的）。编程语言中的其他结构也许有助于我们表示值的结构，但在程序执行期间不能传递，因而是二等公民。

Scala和Groovy等语言的实践已经证明，让方法等概念作为一等值可以扩充程序员的工具库，  
从而让编程变得更容易。一旦程序员熟悉了这个强大的功能，他们就再也不愿意使用没有这一功  
能的语言了。因此， Java 8的设计者决定允许方法作为值，让编程更轻松。此外，让方法作为值也构成了其他若干Java 8功能（如Stream）的基础。

4.2 方法引用

比方说，你想要筛选一个目录中的所有隐藏  
文件。你需要编写一个方法，然后给它一个File，它就会告诉你文件是不是隐藏的。幸好， File  
类里面有一个叫作isHidden的方法。我们可以把它看作一个函数，接受一个File，返回一个布  
尔值。但要用它做筛选，你需要把它包在一个FileFilter对象里，然后传递给File.listFiles  
方法，如下所示：

|  |
| --- |
| File[] hiddenFiles = new File(".").listFiles(new FileFilter() { public boolean accept(File file) {  return file.isHidden();  }  }); |

Java8可以这么写。

File[] hiddenFiles = new File(".").listFiles(File::isHidden);  
你已经有了函数isHidden，因此只需用Java 8的方法引用::语法（即“把这  
个方法作为值”）将其传给listFiles方法；

4.3 Lambda匿名函数

除了允许（命名）函数成为一等值外， Java 8还体现了更广义的将函数作为值的思想，包括  
Lambda①（或匿名函数）。比如，你现在可以写(int x) -> x + 1，表示“调用时给定参数x，  
就返回x + 1值的函数”。

## 5.传递代码的例子

假设你有一个Apple类，它有一个getColor方法，还有一个变量inventory保存着一个Apples的列表。你可能想要选出所有的绿苹果，并返回一个列表。通常我们用筛选（filter）一词来表达这个概念。在Java 8之前，你可能会写这样一个方法filterGreenApples：

|  |
| --- |
| public static List<Apple> filterGreenApples(List<Apple> inventory){ List<Apple> result = new ArrayList<>();  for (Apple apple: inventory){  if ("green".equals(apple.getColor())) { result.add(apple);  } } return result;  } |

但是接下来，有人可能想要选出重的苹果，比如超过150克，于是你心情沉重地写了下面这  
个方法，甚至用了复制粘贴：

|  |
| --- |
| public static List<Apple> filterHeavyApples(List<Apple> inventory){ List<Apple> result = new ArrayList<>(); for (Apple apple: inventory){ if (apple.getWeight() > 150) { result.add(apple); } } return result; } |

Java8代码：

|  |
| --- |
| **public static boolean** isGreenApple(Apple apple) {  **return "green"**.equals(apple.getColor());  } **public static boolean** isHeavyApple(Apple apple) {  **return** apple.getWeight() > 150;} |
| **public static** List<Apple> filterApples(List<Apple> inventory, Predicate<Apple> p){  List<Apple> result = **new** ArrayList<>();  **for**(Apple apple : inventory){  **if**(p.test(apple)){  result.add(apple);  }  }  **return** result; } |

上面方法用法：

|  |
| --- |
| *// [Apple{color='green', weight=80}, Apple{color='green', weight=155}]* List<Apple> greenApples = *filterApples*(inventory, FilteringApples::*isGreenApple*); System.***out***.println(greenApples);  *// [Apple{color='green', weight=155}]* List<Apple> heavyApples = *filterApples*(inventory, FilteringApples::*isHeavyApple*); System.***out***.println(heavyApples); |

Predicate：谓词。

前 面 的 代 码 传 递 了 方 法 Apple::isGreenApple （ 它 接 受 参 数 Apple 并 返 回 一 个boolean）给filterApples，后者则希望接受一个Predicate<Apple>参数。谓词（predicate）在数学上常常用来代表一个类似函数的东西，它接受一个参数值，并返回true或false。你在后面会看到， Java 8也会允许你写Function<Apple,Boolean>——在学校学过函数却没学过谓词的读者对此可能更熟悉，但用Predicate<Apple>是更标准的方式，效率也会更高一点儿，这避免了把boolean封装在Boolean里面。

## 6.Lambda

把方法作为值来传递显然很有用，但要是为类似于isHeavyApple和isGreenApple这种可能只用一两次的短方法写一堆定义有点儿烦人。不过Java 8也解决了这个问题，它引入了一套新记法（匿名函数或Lambda）。

|  |
| --- |
| *// [Apple{color='green', weight=80}, Apple{color='green', weight=155}]* List<Apple> greenApples2 = *filterApples*(inventory, (Apple a) -> **"green"**.equals(a.getColor()));  System.***out***.println(greenApples2);  *// [Apple{color='green', weight=155}]* List<Apple> heavyApples2 = *filterApples*(inventory, (Apple a) -> a.getWeight() > 150);  System.***out***.println(heavyApples2);  *// []* List<Apple> weirdApples = *filterApples*(inventory, (Apple a) -> a.getWeight() < 80 || **"brown"**.equals(a.getColor())); System.***out***.println(weirdApples); |

所以，你甚至都不需要为只用一次的方法写定义；代码更干净、更清晰，因为你用不着去找  
自己到底传递了什么代码。但要是Lambda的长度多于几行（它的行为也不是一目了然）的话，  
那你还是应该用方法引用来指向一个有描述性名称的方法，而不是使用匿名的Lambda。你应该  
以代码的清晰度为准绳

## 7.默认方法

在Java 8里，你现在可以直接对List调用sort方法。它是用Java 8 List接口中如下所  
示的默认方法实现的，它会调用Collections.sort静态方法。

**default** void sort(Comparator<? super E> c) {  
Collections.sort(this, c);

}

## 8.Java8核心思想以及其他思想

Java中从函数式编程中引入的两个核心思想：将方法和Lambda作为一等值，以  
及在没有可变共享状态时，函数或方法可以有效、安全地并行执行。

1. 在Java 8里有一个Optional<T>类，如果你能一致地使用它的话，就可以帮助你避免出现NullPointer异常。它是一个容器对象，可以包含，也可以不包含一个值。 Optional<T>中有方法来明确处理值不存在的情况，这样就可以避免NullPointer异常了。换句话说，它使用类型 系 统 ， 允 许 你 表 明 我 们 知 道 一 个 变 量 可 能 会 没 有 值 。

# 通过行为参数化传递代码

行为参数化就是可以帮助你处理频繁变更的需求的一种软件开发模式。一言以蔽之，它意味  
着拿出一个代码块，把它准备好却不去执行它。这个代码块以后可以被你程序的其他部分调用，  
这意味着你可以推迟这块代码的执行。例如，你可以将代码块作为参数传递给另一个方法，稍后  
再去执行它。这样，这个方法的行为就基于那块代码被参数化了。

## 2.1应对不断变化的需求

2.1.1 筛选绿色苹果

第一个解决方案：

|  |
| --- |
| **public static** List<Apple> filterGreenApples(List<Apple> inventory){  List<Apple> result = **new** ArrayList<>();  **for**(Apple apple: inventory){  **if**(**"green"**.equals(apple.getColor())){  result.add(apple);  }  }  **return** result; } |

突出显示的行就是筛选绿苹果所需的条件。但是现在农民改主意了，他还想要筛选红苹果。你该怎么做呢？简单的解决办法就是复制这个方法，把名字改成filterRedApples，然后更改if条件来匹配红苹果。然而，要是农民想要筛选多种颜色：浅绿色、暗红色、黄色等，这种方法就应付不了了。

2.1.2 把颜色作为参数

|  |
| --- |
| **public static** List<Apple> filterApplesByColor(List<Apple> inventory, String color){  List<Apple> result = **new** ArrayList<>();  **for**(Apple apple: inventory){  **if**(apple.getColor().equals(color)){  result.add(apple);  }  }  **return** result; } |

但是以后，我需要把重量作为参数或者颜色和重量一起筛选呢？难道每次修改代码？

## 2.2 行为参数化

定义一个谓词，和其一系列的实现类

|  |
| --- |
| **interface** ApplePredicate{  **public boolean** test(Apple a); }  **static class** AppleWeightPredicate **implements** ApplePredicate{  **public boolean** test(Apple apple){  **return** apple.getWeight() > 150;   } } **static class** AppleColorPredicate **implements** ApplePredicate{  **public boolean** test(Apple apple){  **return "green"**.equals(apple.getColor());  } }  **static class** AppleRedAndHeavyPredicate **implements** ApplePredicate{  **public boolean** test(Apple apple){  **return "red"**.equals(apple.getColor())   && apple.getWeight() > 150;   } } |

你可以把这些标准看作filter方法的不同行为。你刚做的这些和“策略设计模式” ①相关，  
它让你定义一族算法，把它们封装起来（称为“策略”），然后在运行时选择一个算法。在这里，  
算法族就是ApplePredicate，不同的策略就是AppleHeavyWeightPredicate和AppleGreenColorPredicate。

你需要filterApples方法接受ApplePredicate对象，对Apple做条件测试。这就是行为参数化：让方法接受多种行为（或战略）作为参数，并在内部使用，来完成不同的行为。

根据抽象条件筛选苹果

|  |
| --- |
| **public static** List<Apple> filter(List<Apple> inventory, ApplePredicate p){  List<Apple> result = **new** ArrayList<>();  **for**(Apple apple : inventory){  **if**(p.test(apple)){  result.add(apple);  }  }  **return** result; } |

这段代码比我们第一次尝试的时候灵活多了，读起来、用起来也更容易！现在你可以创建不同的ApplePredicate对象，并将它们传递给filterApples方法。

List<Apple> redAndHeavyApples =filterApples(inventory, new AppleRedAndHeavyPredicate());

正如我们先前解释的那样，行为参数化的好处在于你可以把迭代要筛选的集合的逻辑与对集合中每个元素应用的行为区分开来。这样你可以重复使用同一个方法，给它不同的行为来达到不同的目的。

## 2.3 对付啰嗦

目前，当要把新的行为传递给filterApples方法的时候，你不得不声明好几个实现ApplePredicate接口的类，然后实例化好几个只会提到一次的ApplePredicate对象。下面的程序总结了你目前看到的一切。这真是很啰嗦，很费时间！

费这么大劲儿真没必要，能不能做得更好呢？ Java有一个机制称为匿名类，它可以让你同时声明和实例化一个类。它可以帮助你进一步改善代码，让它变得更简洁。

使用匿名类过滤苹果

|  |
| --- |
| *// [Apple{color='red', weight=120}]* List<Apple> redApples2 = *filter*(inventory, **new** ApplePredicate() {  **public boolean** test(Apple a){  **return** a.getColor().equals(**"red"**);   } }); System.***out***.println(redApples2); |

但匿名类还是不够好。第一，它往往很笨重，因为它占用了很多空间。还拿前面的例子来看，  
如下面高亮的代码所示：

使用Lambda表达式解决啰嗦：

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

将List抽象化

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

|  |
| --- |
| public interface Predicate<T>{ boolean test(T t); } public static <T> List<T> filter(List<T> list, Predicate<T> p){ List<T> result = new ArrayList<>(); for(T e: list){ if(p.test(e)){ result.add(e); } } return result; } |

现在你可以把filter方法用在香蕉、桔子、 Integer或是String的列表上了。这里有一个使用Lambda表达式的例子：

|  |
| --- |
| List<Apple> redApples = filter(inventory, (Apple apple) -> "red".equals(apple.getColor())); List<Integer> evenNumbers = filter(numbers, (Integer i) -> i % 2 == 0); |

## 2.4 真实的例子

2.4 .1 用Comparator 排序

在Java 8中， List自带了一个sort方法（你也可以使用Collections.sort）。 sort的行为可以用java.util.Comparator对象来参数化，它的接口如下

|  |
| --- |
| // java.util.Comparator public interface Comparator<T> {  public int compare(T o1, T o2); } |

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

|  |
| --- |
| inventory.sort(new Comparator<Apple>() { public int compare(Apple a1, Apple a2){  return a1.getWeight().compareTo(a2.getWeight()); }  });  用lambda:  inventory.sort( (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())); |

2.4.2 用runnable 执行代码块

|  |
| --- |
| 之前： public interface Runnable{  public void run(); }  Thread t = new Thread(new Runnable() { public void run(){  System.out.println("Hello world"); }  });  Lambda：  Thread t = new Thread(() -> System.out.println("Hello world")); |

2.4.3 GUI事件

忽略。

# Lambda表达式

Lambda表达式，它可以让你很简洁地表示一个行为或传递代码。现在你可以把Lambda  
表达式看作匿名功能，它基本上就是没有声明名称的方法，但和匿名类一样，它也可以作为参  
数传递给一个方法。

## 3.1 Lambda管中窥豹

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

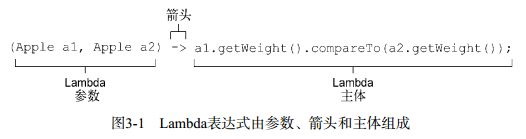
匿名——我们说匿名，是因为它不像普通的方法那样有一个明确的名称：写得少而想  
得多！

函数——我们说它是函数，是因为Lambda函数不像方法那样属于某个特定的类。但和方  
法一样， Lambda有参数列表、函数主体、返回类型，还可能有可以抛出的异常列表。

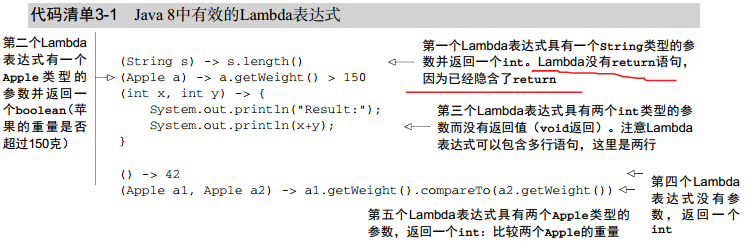
传递——Lambda表达式可以作为参数传递给方法或存储在变量中。

简洁——无需像匿名类那样写很多模板代码。

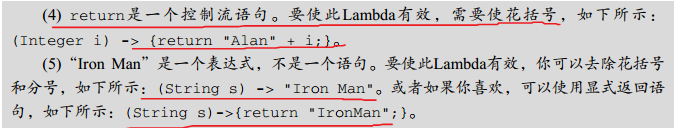
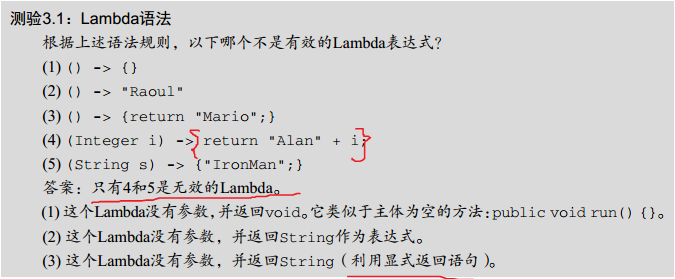
它可以让你十分简明地传递代码。理论上来说，你在Java 8之前做不了的事情， Lambda也做不了。但是，现在你用不着再用匿名类写一堆笨重的代码，来体验行为参数化的好处了！



5个Lambda表达式例子：



测验：



## 3.2 在哪里以及如何使用Lambda

3.2.1 函数式接口

只有在需要函数式接口的时候才可以传递Lambda。

函数式接口就是只定义一个抽象方法的接口，如我们在第2章中谈到的Comparator和Runnable。

|  |
| --- |
| public interface Comparator<T> { int compare(T o1, T o2);  } public interface Runnable{ void run(); } public interface ActionListener extends EventListener{ void actionPerformed(ActionEvent e); } public interface Callable<V>{ V call(); } public interface PrivilegedAction<V>{ V run();} |

接口现在还可以拥有默认方法（即在类没有对方法进行实现时，其主体为方法提供默认实现的方法）。哪怕有很多默认方法，只要接口只定义了一个抽象方法，它就仍然是一个函数式接口。

用函数式接口可以干什么呢？ Lambda表达式允许你直接以内联的形式为函数式接口的抽象方法提供实现， 并把整个表达式作为函数式接口的实例（具体说来，是函数式接口一个具体实现的实例）。你用匿名内部类也可以完成同样的事情，只不过比较笨拙：需要提供一个实现，然后再直接内联将它实例化。

3.2.2 函数描述符FunctionalInterface

函数式接口的抽象方法的签名基本上就是Lambda表达式的签名。我们将这种抽象方法叫作  
函数描述符。 例如， Runnable接口可以看作一个什么也不接受什么也不返回（void）的函数的签名，因为它只有一个叫作run的抽象方法，这个方法什么也不接受，什么也不返回（void）。

**@FunctionalInterface**又是怎么回事？  
如果你去看看新的Java API，会发现函数式接口带有@FunctionalInterface的标注（3.4  
节中会深入研究函数式接口，并会给出一个长长的列表）。这个标注用于表示该接口会设计成  
一个函数式接口。如果你用@FunctionalInterface定义了一个接口，而它却不是函数式接  
口的话，编译器将返回一个提示原因的错误。例如，错误消息可能是“Multiple non-overriding  
abstract methods found in interface Foo”，表明存在多个抽象方法。请注意，@FunctionalInterface不是必需的，但对于为此设计的接口而言，使用它是比较好的做法。 它就像是@Override  
标注表示方法被重写了。

## 3.3 把Lambda付诸实践：环绕执行模式

让我们通过一个例子，看看在实践中如何利用Lambda和行为参数化来让代码更为灵活，更  
为简洁。资源处理（例如处理文件或数据库）时一个常见的模式就是打开一个资源，做一些处理，  
然后关闭资源。这个设置和清理阶段总是很类似，并且会围绕着执行处理的那些重要代码。这就  
是所谓的环绕执行（execute around） 模式。

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

}//带资源的try,简化了代码。

3.3.1 记得行为参数化

现在这段代码是有局限的。你只能读文件的第一行。如果你想要返回头两行，甚至是返回使  
用最频繁的词，该怎么办呢？你需要一种方法把行为传递给processFile，以便它可以利用  
BufferedReader执行不同的行为。传递行为正是Lambda的拿手好戏。那要是想一次读两行，这个新的processFile方法看起来又该是什么样的呢？基本上，你需要一个接收BufferedReader并返回String的Lambda。例如，下面就是从BufferedReader中打印两行的写法.

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

3.3.2 使用函数式接口来传递行为

我们前面解释过了， Lambda仅可用于上下文是函数式接口的情况。你需要创建一个能匹配BufferedReader -> String，还可以抛出IOException异常的接口。

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

3.3.3 第三部，执行一个行为

任何BufferedReader -> String形式的Lambda都可以作为参数来传递，因为它们符合BufferedReaderProcessor接口中定义的process方法的签名。现在你只需要一种方法在processFile主体内执行Lambda所代表的代码。请记住， Lambda表达式允许你直接内联，为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。因此，你可以在processFile主体内，对得到的BufferedReaderProcessor对象调用process方法执行处理。

|  |
| --- |
| 使用：  **public static** String processFile(BufferedReaderProcessor p) **throws** IOException {  **try**(BufferedReader br = **new** BufferedReader(**new** FileReader(**"lambdasinaction/chap3/data.txt"**))){  **return** p.process(br);//处理BufferReader对象  } } |

3.3.4 第四步：传递Lambda

|  |
| --- |
| String oneLine = *processFile*((BufferedReader b) -> b.readLine()); System.***out***.println(oneLine);  String twoLines = *processFile*((BufferedReader b) -> b.readLine() + b.readLine()); System.***out***.println(twoLines); |



我们已经展示了如何利用函数式接口来传递Lambda，但你还是得定义你自己的接口。在下  
一节中，我们会探讨Java 8中加入的新接口，你可以重用它来传递多个不同的Lambda。

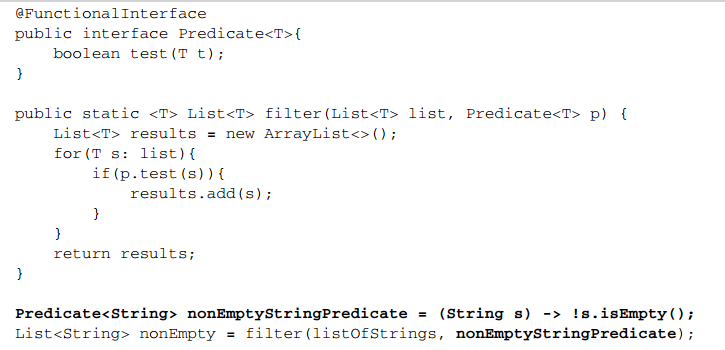
## 3.4 使用函数式接口

函数式接口定义且只定义了一个抽象方法。函数式接口很有用，因为抽象方法的签名可以描述Lambda表达式的签名。函数式接口的抽象方法的签名称为函数描述符。所以为了应用不同的Lambda表达式，你需要一套能够描述常见函数描述符的函数式接口。Java API中已经有了几个函数式接口，比如你在3.2节中见到的Comparable、 Runnable和Callable。

Java 8的库设计师帮你在java.util.function包中引入了几个新的函数式接口。我们接下来会介绍Predicate、 Consumer和Function.

3.4.1 Predicate

java.util.function.Predicate<T>接口定义了一个名叫test的抽象方法，它接受泛型T对象，并返回一个boolean。这恰恰和你先前创建的一样，现在就可以直接使用了。在你需要表示一个涉及类型T的布尔表达式时，就可以使用这个接口。



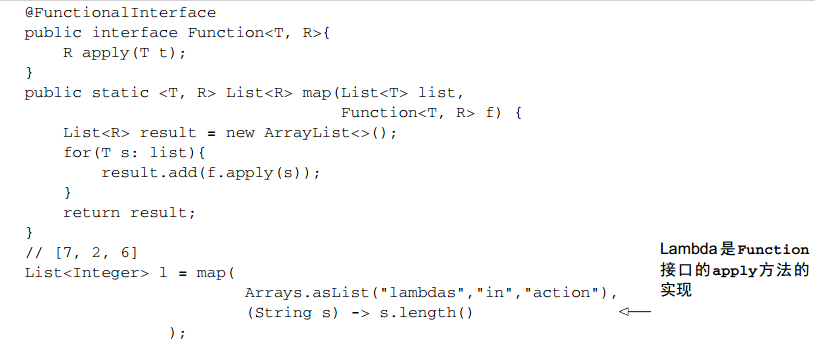
3.4.2 Consumer

java.util.function.Consumer<T>定义了一个名叫accept的抽象方法，它接受泛型T的对象，没有返回（void）。你如果需要访问类型T的对象，并对其执行某些操作，就可以使用这个接口。比如，你可以用它来创建一个forEach方法，接受一个Integers的列表，并对其中每个元素执行操作。在下面的代码中，你就可以使用这个forEach方法，并配合Lambda来打印列表中的所有元素。

|  |
| --- |
| @FunctionalInterface **public interface** Consumer<T>{  **void** accept(T t); } **public static** <T> **void** forEach(List<T> list, Consumer<T> c){  **for**(T i: list){  c.accept(i);  } } forEach(  Arrays.asList(1,2,3,4,5),  (Integer i) -> System.***out***.println(**i**)// Lambda是**Consumer**中  **accept**方法的实现  ); |

3.4.3 Function

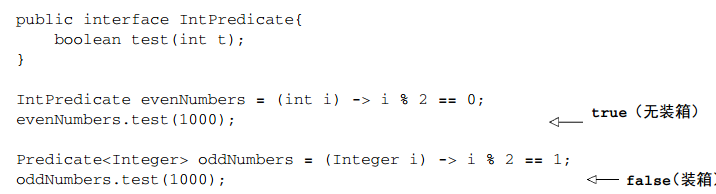
java.util.function.Function<T, R>接口定义了一个叫作apply的方法，它接受一个泛型T的对象，并返回一个泛型R的对象。如果你需要定义一个Lambda，将输入对象的信息映射到输出，就可以使用这个接口（比如提取苹果的重量，或把字符串映射为它的长度）。在下面的代码中，我们向你展示如何利用它来创建一个map方法，以将一个String列表映射到包含每个String长度的Integer列表。



**原始类型特化**

基本类型的自动装箱拆箱性能是要付出代价的，装箱后的值本质上就是把原始类型包裹起来，并保存在堆里。因此，装箱后的值需要更多的内存，并需要额外的内存搜索来获取被包裹的原始值。

Java 8为我们前面所说的函数式接口带来了一个专门的版本，以便在输入和输出都是原始类  
型时避免自动装箱的操作。比如，在下面的代码中，使用IntPredicate就避免了对值1000进行装箱操作，但要是用Predicate<Integer>就会把参数1000装箱到一个Integer对象中：



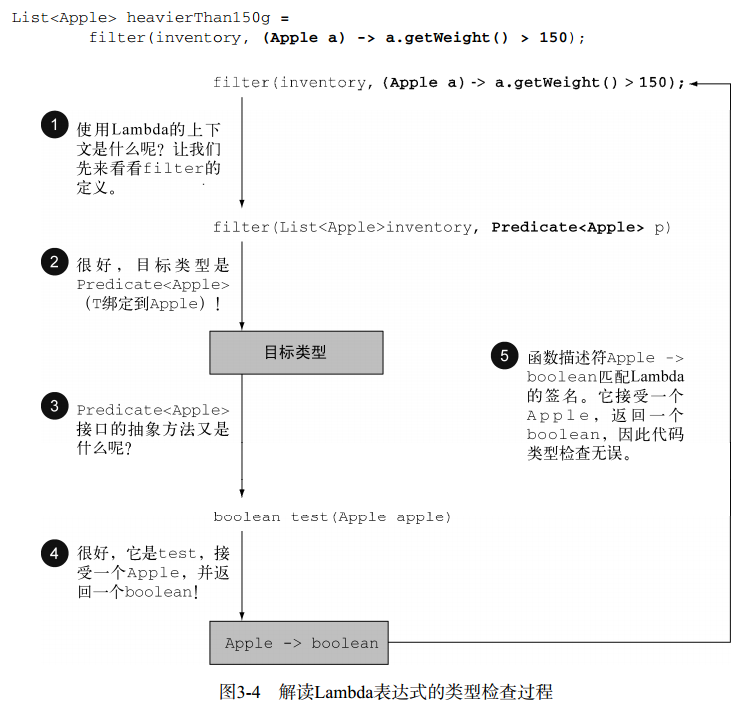
一般来说，针对专门的输入参数类型的函数式接口的名称都要加上对应的原始类型前缀，比  
如DoublePredicate、 IntConsumer、 LongBinaryOperator、 IntFunction等。 Function接口还有针对输出参数类型的变种： ToIntFunction<T>、 IntToDoubleFunction等。

## 3.5类型检查、类型推断以及限制

当我们第一次提到Lambda表达式时，说它可以为函数式接口生成一个实例。然而， Lambda  
表达式本身并不包含它在实现哪个函数式接口的信息。为了全面了解Lambda表达式，你应该知  
道Lambda的实际类型是什么。

3.5.1 类型检查

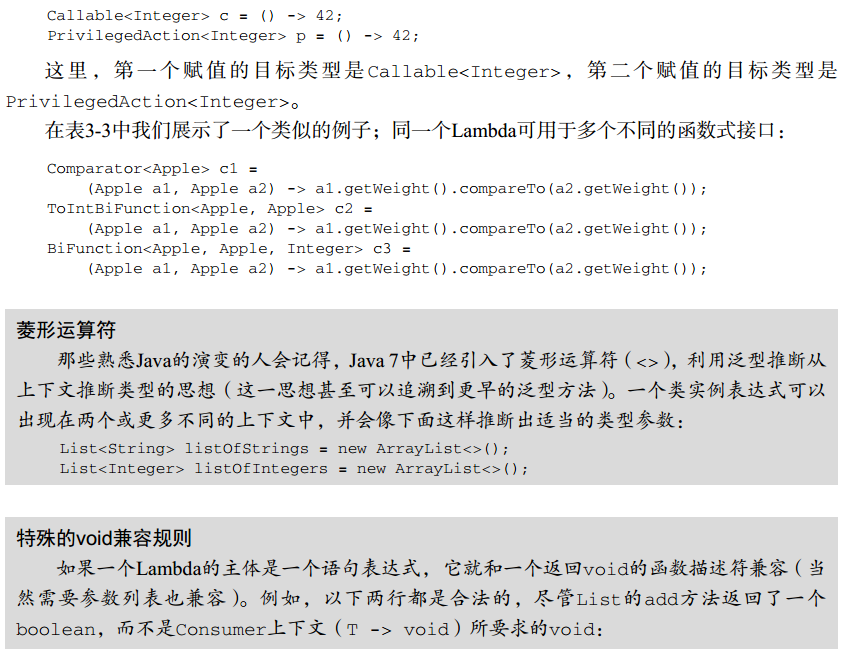
Lambda的类型是从使用Lambda的上下文推断出来的。上下文（比如，接受它传递的方法的参数，或接受它的值的局部变量）中Lambda表达式需要的类型称为目标类型。

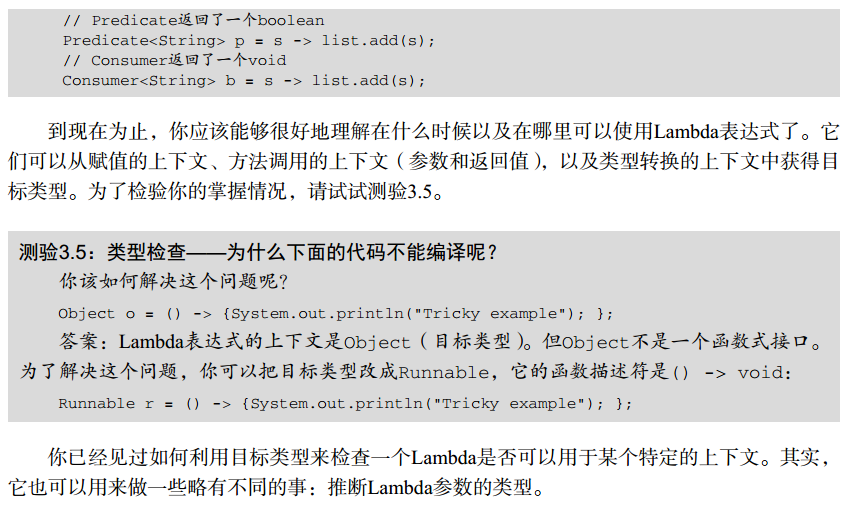


θ 首先，你要找出filter方法的声明。  
θ 第二，要求它是Predicate<Apple>（目标类型）对象的第二个正式参数。  
θ 第三， Predicate<Apple>是一个函数式接口，定义了一个叫作test的抽象方法。  
θ 第四， test方法描述了一个函数描述符，它可以接受一个Apple，并返回一个boolean。  
θ 最后， filter的任何实际参数都必须匹配这个要求。

3.5.2 同样的Lambda，不同的函数式接口

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





3.5.3 类型推断

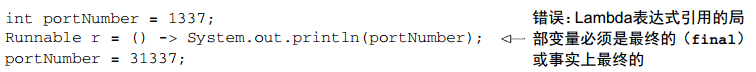


3.5.4 使用局部变量

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

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

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



**对局部变量的限制**

为什么局部变量有这些限制。

1. 实例变量和局部变量背后的实现有一个关键不同。实例变量都存储在堆中，而局部变量则保存在栈上。如果Lambda可以直接访问局部变量，而且Lambda是在一个线程中使用的，则使用Lambda的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量。因此， Java在访问自由局部变量时，实际上是在访问它的副本，而不是访问原始变量。如果局部变量仅仅赋值一次那就没有什么区别了——因此就有了这个限制。
2. 这一限制不鼓励你使用改变外部变量的典型命令式编程模式（我们会在以后的各章中解释，这种模式会阻碍很容易做到的并行处理）。

**闭包**  
你可能已经听说过闭包（closure，不要和Clojure编程语言混淆）这个词，你可能会想  
Lambda是否满足闭包的定义。用科学的说法来说，闭包就是一个函数的实例，且它可以无限  
制地访问那个函数的非本地变量。例如，闭包可以作为参数传递给另一个函数。它也可以访  
问和修改其作用域之外的变量。现在， Java 8的Lambda和匿名类可以做类似于闭包的事情：  
它们可以作为参数传递给方法，并且可以访问其作用域之外的变量。但有一个限制：它们不  
能修改定义Lambda的方法的局部变量的内容。这些变量必须是隐式最终的。可以认为Lambda  
是对值封闭，而不是对变量封闭。如前所述，这种限制存在的原因在于局部变量保存在栈上，  
并且隐式表示它们仅限于其所在线程。如果允许捕获可改变的局部变量，就会引发造成线程  
不安全的新的可能性，而这是我们不想看到的（实例变量可以，因为它们保存在堆中，而堆  
是在线程之间共享的）

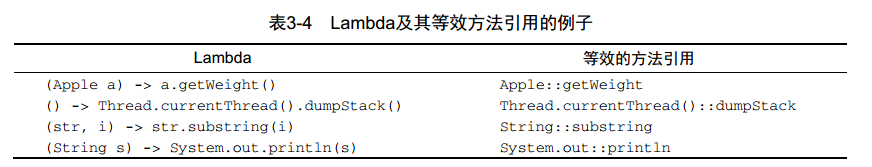
## 3.6 方法引用

方法引用。可以把它们视为某些Lambda的快捷写法。在一些情况下，比起使用Lambda表达式，它们似乎更易读，感觉也更自然:

|  |
| --- |
| Lambda:  inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight())); System.***out***.println(inventory);  方法引用：  inventory.sort(*comparing*(Apple::getWeight)); System.***out***.println(inventory); |

3.6.1 管中窥豹

方法引用可以被看作仅仅调用特定方法的Lambda的一种快捷写法。它的基本思想是，如果一个Lambda代表的只是“直接调用这个方法”，那最好还是用名称来调用它，而不是去描述如何调用它。事实上，方法引用就是让你根据已有的方法实现来创建Lambda表达式。但是，显式地指明方法的名称，你的代码的可读性会更好。当 你 需 要使用 方 法 引用时 ， 目 标引用 放 在 分隔符::前 ，方法 的 名 称放在 后 面 。



**如何构建方法引用**

方法引用主要有三类。

1. 指向静态方法的方法引用（例如Integer的parseInt方法，写作Integer::parseInt）。

(2) 指 向 任 意 类 型 实 例 方 法 的 方 法 引 用 （ 例 如 String 的 length 方 法 ， 写 作String::length）。

(3) 指向现有对象的实例方法的方法引用（假设你有一个局部变量expensiveTransaction  
用于存放Transaction类型的对象，它支持实例方法getValue，那么你就可以写expensiveTransaction::getValue）。

第二种和第三种方法引用可能乍看起来有点儿晕。类似于String::length的第二种方法引用的思想就是你在引用一个对象的方法，而这个对象本身是Lambda的一个参数。例如， Lambda表达式(String s) -> s.toUppeCase()可以写作String::toUpperCase。但第三种方法引用指 的 是 ， 你 在 Lambda 中 调 用 一 个 已 经 存 在 的 外 部 对 象 中 的 方 法 。 例 如 ， Lambda 表 达 式  
()->expensiveTransaction.getValue()

可以写作

expensiveTransaction::getValue。

还有针对**构造函数、数组构造函数和父类调用**（super-call）的一些特殊形式的方法  
引用。

想要对一个字符串的List排序，忽略大小写。 List的sort方法需要一个Comparator作为参数。

|  |
| --- |
| List<String> str = Arrays.asList("a","b","A","B"); str.sort((s1, s2) -> s1.compareToIgnoreCase(s2)); |

3.6.2 构造函数引用

对于一个现有构造函数，你可以利用它的名称和关键字new来创建它的一个引用：ClassName::new。它的功能与指向静态方法的引用类似。例如，假设有一个构造函数没有参数。它适合Supplier的签名() -> Apple。你可以这样做：

Supplier<Apple> c1 = Apple::new;  
Apple a1 = c1.get();

这就等价于：

Supplier<Apple> c1 = () -> new Apple();  
Apple a1 = c1.get();

如果你的构造函数的签名是Apple(Integer weight)，那么它就适合Function接口的签名，于是你可以这样写：

Function<Integer, Apple> c2 = Apple::new;  
Apple a2 = c2.apply(110);

这就等价于：

|  |
| --- |
| Function<Integer, Apple> c2 = (weight) -> new Apple(weight);  Apple a2 = C2.apply(110); |

在下面的代码中，一个由Integer构成的List中的每个元素都通过我们前面定义的类似的map方法传递给了Apple的构造函数，得到了一个具有不同重量苹果的List：

|  |
| --- |
| List<Integer> **weights** = Arrays.*asList*(7, 3, 4, 10); List<Apple> **apples** = *map*(**weights**, Apple::**new**); **public static** List<Apple> map(List<Integer> list, Function<Integer, Apple> f){  List<Apple> result = **new** ArrayList<>();  **for**(Integer e: list){  result.add(f.apply(e));  }  **return** result; } |

如果你有一个具有两个参数的构造函数Apple(String color, Integer weight)，那么它就适合BiFunction接口的签名，于是你可以这样写：

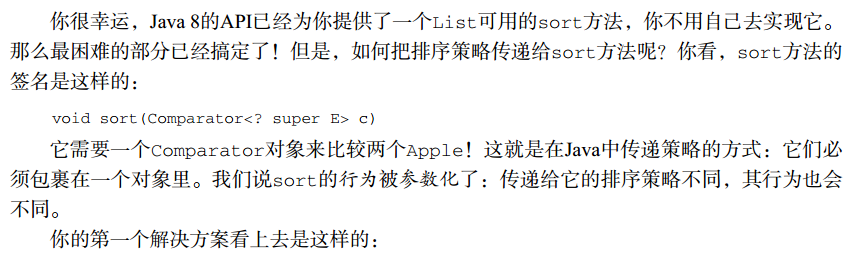
|  |
| --- |
| BiFunction<String, Integer, Apple> **c3** = Apple::**new**;  Apple **c3** = **c3**.apply(**"green"**, 110); BiFunction<String, Integer, Apple> **c3** = (color, weight) -> **new** Apple(color, weight); Apple **c3** = **c3**.apply(**"green"**, 110); |

不将构造函数实例化却能够引用它，这个功能有一些有趣的应用。例如，你可以使用Map来将构造函数映射到字符串值。你可以创建一个giveMeFruit方法，给它一个String和一个Integer，它就可以创建出不同重量的各种水果：

|  |
| --- |
| **static** Map<String, Function<Integer, Fruit>> *map* = **new** HashMap<>(); **static** {  *map*.put(**"apple"**, Apple::**new**);  *map*.put(**"orange"**, Orange::**new**);  *// etc...* } **public static** Fruit giveMeFruit(String fruit, Integer weight){  **return** *map*.get(fruit.toLowerCase())  .apply(weight); } |

## 3.7 Lambda和方法引用实战

3.7.1 第一步 传递代码



|  |
| --- |
| **static class** AppleComparator **implements** Comparator<Apple> {  **public int** compare(Apple a1, Apple a2){  **return** a1.getWeight().compareTo(a2.getWeight());  } } inventory.sort(**new** AppleComparator()); System.**out**.println(inventory); |

3.7.2 第二步 使用匿名类

你在前面看到了，你可以使用匿名类来改进解决方案，而不是实现一个Comparator却只实  
例化一次：

|  |
| --- |
| inventory.sort(**new** Comparator<Apple>() {  **public int** compare(Apple a1, Apple a2){  **return** a1.getWeight().compareTo(a2.getWeight());  }}); System.***out***.println(inventory); |

3.7.3 第三步 使用Lambda表达式

但你的解决方案仍然挺啰嗦的。 Java 8引入了Lambda表达式，它提供了一种轻量级语法来实  
现相同的目标： 传递代码。你看到了，在需要函数式接口的地方可以使用Lambda表达式。我们回顾一下：函数式接口就是仅仅定义一个抽象方法的接口。抽象方法的签名（称为函数描述符）  
描述了Lambda表达式的签名。在这个例子里， Comparator代表了函数描述符(T, T) -> int。因为你用的是苹果，所以它具体代表的就是(Apple, Apple) -> int。改进后的新解决方案看上去就是这样的了。

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

我们前面解释过了， Java编译器可以根据Lambda出现的上下文来推断Lambda表达式参数的  
类型。那么你的解决方案就可以重写成这样：

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

你的代码还能变得更易读一点吗？ Comparator具有一个叫作comparing的静态辅助方法，它可以接受一个Function来提取Comparable键值，并生成一个Comparator对象（我们会在第9章解释为什么接口可以有静态方法）。它可以像下面这样用（注意你现在传递的Lambda只有一个参数： Lambda说明了如何从苹果中提取需要比较的键值）

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

现在你可以把代码再改得紧凑一点了：

import static java.util.Comparator.comparing;  
inventory.sort(comparing((a) -> a.getWeight()));

3.7.4 使用方法引用

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

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

## 3.8 复合Lambda表达式的有用方法

Java 8的好几个函数式接口都有为方便而设计的方法。具体而言，许多函数式接口，比如用  
于传递Lambda表达式的Comparator、Function和Predicate都提供了允许你进行复合的方法。这是什么意思呢？在实践中，这意味着你可以把多个简单的Lambda复合成复杂的表达式。比如，你可以让两个谓词之间做一个or操作，组合成一个更大的谓词。而且，你还可以让一个函数的结果成为另一个函数的输入。你可能会想，函数式接口中怎么可能有更多的方法呢？（毕竟，这违背了函数式接口的定义啊！）窍门在于，我们即将介绍的方法都是默认方法，也就是说它们不是抽象方法。

3.8.1 比较器复合

我们前面看到，你可以使用静态方法Comparator.comparing，根据提取用于比较的键值  
的Function来返回一个Comparator，如下所示：

Comparator<Apple> c = Comparator.comparing(Apple::getWeight);

**1.逆序**

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

**2.比较器链**