

# 第1章 lambda表达式

* 1.1：为什么要使用lambda表达式
* 1.2：lambda表达式的语法
* 1.3：函数式接口
* 1.4：方法引用
* 1.5：构造器引用
* 1.6：变量作用域
* 1.7：默认方法
* 1.8：接口中的静态方法

Java作为一门面向对象的编程语言诞生于20世纪90年代，在当时，面向对象编程是软件开发的主流模式。在面向对象编程出现之前，也曾诞生过像Lisp和Scheme这样的函数式编程语言，但它们只活跃于学术圈中。最近，由于在并发和事件驱动（或者称“互动”）编程中的优势，函数式编程又逐渐变得重要起来。这并不意味着面向对象编程不好，相反，最终的趋势是将面向对象编程和函数式编程结合起来。即使你对并发等功能不感兴趣，函数式编程也会给你带来帮助。例如，如果语言有了非常方便的函数表达式语法，集合API就会变得异常强大。

Java8主要是在原来面向对象的基础上增加了函数式编程的能力。在本章中，你将学习基本的语法。下一章将会向你介绍如何利用这些语法来使用Java集合类，第3章将介绍如何构建自己的函数式AP I。

本章的要点包括：

* 一个lambda表达式是一个带有参数的代码块。
* 当你想要代码块在以后某个时间点执行时，可以使用lambda表达式。
* lambda表达式可以被转换为函数式接口。
* lambda表达式可以在闭包作用域中有效地访问final变量。
* 方法和构造器引用可以引用方法或构造器，但无须调用它们。
* 你现在可以向接口添加默认（default）和静态（static）方法来提供具体的实现。
* 你必须解决接口中多个默认方法之间的冲突

## 1.1 为什么要使用lambda表达式

“lambda表达式”是一段可以传递的代码，因此它可以被执行一次或多次。在学习语法（甚至包括一些奇怪的术语）之前，我们先回顾一下之前在Java中一直使用的相似的代码块。

当我们要在另一个独立线程中执行一些逻辑时，通常会将代码放在一个实现

Runnable接口的类的run方法中，如下所示：

|  |
| --- |
| **public class** \_00\_Worker **implements** Runnable {  @Override  **public void** run() {  **for** (**int** i = 0; i < 1000; i++) {  *//doWork();* }  } } |

然后，当我们希望执行这段代码时，会构造一个Worker类的实例。然后将该实例提交到一个线程池中，或者简单点，直接启动一个新的线程：

|  |
| --- |
| \_00\_Worker worker = **new** \_00\_Worker(); **new** Thread(worker).start(); |

这段代码的关键在于，run方法中包含了你希望在另一个线程中执行的代码。我们考虑一下用一个自定义的比较器来进行排序。如果你希望按照字符串的长度而不是默认的字典顺序来排序，那么你可以将一个Comparator对象传递给sort方法：

|  |
| --- |
| **public class** \_02\_LengthComparator **implements** Comparator<String> {  @Override  **public int** compare(String first, String second) {  **return** Integer.*compare*(first.length(), second.length());  }  **public static void** main(String[] args) {  String[] strings = {**"a"**, **"bb"**, **"ccc"**};  Arrays.*sort*(strings, **new** \_02\_LengthComparator());  } } |

sort方法会一直调用compare方法，对顺序不对的元素进行重新排序，直到数组完全有序为止。你给sort方法传递了一段需要比较元素的代码片段，而该代码会被整合到排序逻辑中，而你可能井不关心如何在那里实现。

* 注意：

1. 如果x=y，那么Intege.compare(x,y)会返回0；
2. 如果以x<y它会返回一个负数；
3. 如果x>y，则会返回一个正数。这个静态方法已经被添加到Java7中（请参考第9章）。
4. 还要注意，不应该使用x-y来比较x和y的大小，因为对于大的、符号相反的操作数，这种计算有可能会产生溢出。

按钮回调是另外一个会延迟执行的例子。你将回调操作放到一个实现了监昕器接口的类的某个方法中，然后构造一个实例，并将实例注册到按钮上。在这种情况下，许多开发人员都会使用“匿名类的匿名实例”的方法：

|  |
| --- |
| **public class** \_03\_ButtonEvent **extends** Application {  @Override  **public void** start(Stage primaryStage) {  primaryStage.setTitle(**"Hello World !"**);  Button button = **new** Button();  button.setText(**"hello"**);  button.setOnAction(**new** EventHandler<ActionEvent>() {  @Override  **public void** handle(ActionEvent event) {  System.***out***.println(**"Thinks for clickig!"**);  }  });  StackPane root = **new** StackPane();  root.getChildren().add(button);  Scene scene = **new** Scene(root, 300, 250);  primaryStage.setScene(scene);  primaryStage.show();  }  **public static void** main(String[] args) {  *launch*(args);  } } |

这里的关键是代码处于handle方法中。该代码会在按钮被点击时执行。

* 注意：

由于Java8将JavaFX作为SwingGUI的下一任继承者，我会在这些示例中使用JavaFX（请参考第4章来了解更多关于JavaFX的信息）。当然，细节并不重要，因为不管是Swing、JavaFX还是Android，你都需要为按钮添加一些代码，以使它们在按钮被点击时可以执行。

在所有三个例子中，你会看到相同的方式。一段代码会被传递给其他调用者：也许是一个线程池、一个排序方法，或者是一个按钮。这段代码会在稍后被调用。

到现在为止，在Java中向其他代码传递一段代码并不是很容易。你不可能将代码块到处传递。由于Java是一个面向对象的语言，因此你不得不构建一个属于某个类的对象，由它的某个方法来包含所需的代码。

在其他一些语言中可以直接使用代码块。在很长一段时间里，Java设计者们都拒绝加入这一特性。毕竟，Java的一大优势在于它的简单和一致性。如果一个语言包含了所有可以略微简化代码的特性，那么它就会变得不可维护。但是，在其他那些语言中，并不只是产生线程或者注册按钮点击事件的代码变得更简单了，它们大量的APT都是更简单、更一致、更强大的。虽然我们己经通过类、对象的方式在Java中实现了相似的API和功能，但是这些APT使用起来并不让人感到轻松和愉快。

最近的一段时间，争论的问题己经不再是Java是否要变成一门函数式编程语言，而是如何实现这种改变了。在设计出一个适合Java的解决办法之前已经进行了多年的实验。在下一节中，你将会看到如何在Java8中使用代码块。

## 1.2 lambda表达式的语法

还以上一节中的排序为例。我们传递代码来检查某个字符串的长度是否小于另一个字符串的长度，如下所示：

|  |
| --- |
| Integer.*compare*(first.length(),second.length()) |

first和second是什么呢？它们都是字符串。Java是一个强类型的语言，因此我们必须同时指定类型，如下：

|  |
| --- |
| (String first,String second)->Integer.compare(first.length(),second.length()) |

这就是你见到的第一个“lambda表达式”。这个表达式不仅是一个简单的代码块，还指定了必须传递给代码的所有变量。

为什么要叫这个名字呢？许多年前，在计算机出现之前，有位名叫AlonzoChurch的逻辑学家，他想要证明什么样的数学函数是可以有效计算的。（奇怪的是，当时己经存在了许多己知的函数，但是没有人知道怎样去计算它们的值。）他使用希腊字母的lambda(λ)来标记参数。如果他懂JavaAPI的话，他应该会写下如下代码：

|  |
| --- |
| λfirst.λsecond.Integer.compare(first.length(),second.length()) |

注意：

为什么使用字母λ？难道Church没有其他拉丁字母可用了吗？

事实上，经典的（数学原理）中使用符号^表示自由变量，这启发Church使用大写的lambda来表示参数。但是最终，他选择换回到小写版本。于是从那时起，带有参数变量的表达式都被称为lambda表达式。

你己经见到了Java中lambda表达式的格式：参数、箭头，->，以及一个表达式。如果负责计算的代码无法用一个表达式表示，那么可以用编写方法的方式来编写：即用｛｝包裹代码并明确使用return语句，例如：

|  |
| --- |
| (String first, String second) ->{  if (first.length() < second.length()) return -1;  else if (first.length() > second.length()) return 1;  else return 0;  } |

如果lambda表达式没有参数，你仍可以提供一对空的小括号，如同不含参数的方法那样：

|  |
| --- |
| ( )->{**for**(inti=0;i<1000;i++) doWork();} |

如果一个lambda表达式的参数类型是可以被推导的，那么就可以省略它们的类型，例如：

|  |
| --- |
| Comparator<String> comp =(first,second）*//同（String first,String second）一样* ->Integer.*compare*(first.length(),second.length()); |

这里，编译器会推导出first和second必须是字符串，因为lambda表达式被赋给了一个字符串比较器（我们将会在下一节详细讲解该赋值过程）。

如果某个方法只含有一个参数，并且该参数的类型可以被推导出来，你甚至可以省略小括号：

|  |
| --- |
| EventHandler<ActionEvent>listener=event->System.***out***.println（**"Thanks for clicking！"**）; *//无须（event）→或（ActionEventevent)->* |

* 注意：

你可以像对待方法参数样向lambda表达式的参数添加注解或者final修饰符，如下。

(finalStringname)->。。。

(@NonNullStringname)->。。。

永远不需要为一个lambda表达式执行返回类型，它总是会从上下文中被推导出来。例如，表达式

|  |
| --- |
| (String first,String second)->Integer.*compare*(first.length(),second.length()) |

可以被使用在期望结果类型为int的上下文中。

* 注意：

在lambda表达式中，只在某些分支中返回值（其他分支没有返回值）是不合法的。例如，

（int x)->{ if ( x >= 0 ) return 1 ;｝是不合法的。

## 1.3 函数式接口

正如我们讨论所述，在Java中有许多己有的接口都需要封装代码块，例如Runnable

或者Comparator。lambda表达式与这些接口是向后兼容的。对于只包含一个抽象方法的接口，你可以通过lambda表达式来创建该接口的对象。这种接口被称为函数式接口。

* 注意：

你可能奇怪为什么函数式接口必须只有一个抽象方法。难道接口中的方法不都是抽象的吗？事实上，接口经常会重新声明Object类中的方法，例如toString或者clone，而这些方法声明并不是抽象的。（JavaAPI中的某些接口重新声明Object类的方法，是为了关联javadoc的注释。具体例子可以参考ComparatorAPI。）你将在第1.7节看到更重要的点，即在Java8中接口可以声明非抽象的方法。

为了演示函数式接口转换，我们以Arrays.sort方法为例。该方法的第二个参数需要一个Comparator接口（该接口只含有一个方法）的实例。接下来我们编写一个简单的lambda表达式：

|  |
| --- |
| Arrays.*sort*{words, (first,second)->Integer.*compare*(first.length(),second.length())}; |

在这个表达式背后，Arrays.sort方法会接收一个实现了Comparator<String>接口的类的实例。调用该对象的compare方法会执行lambda表达式中的代码。这些对象和类的管理完全依赖于如何实现，因此比传统的内部类效率更高。你最好将一个lambda表达式想象成一个函数，而不是一个对象，并记住它可以被转换为一个函数式接口。

这种到接口的转换使得lambda表达式非常引人注目，它的语法是如此精简。下面是另外一个示例：

|  |
| --- |
| button.setOnAction(event-> System.***out***.println(**"Thanks for clicking！"**)); |

显然其可读性也比内部类好了很多。

事实上，函数式接口的转换是你在Java中使用lambda表达式能做的唯一一件事。在其他支持函数文本的编程语言中，你可以声明像（String,String)->int这样的函数类型，声明这种类型的变量，并使用这些变量来保存函数表达式。但是，Java设计者们还是决定坚持使用熟悉的接口概念，而没有将函数类型添加到Java中。

* 注意：

你甚至不能将一个lambda表达式赋值给一个Object类型的变量，因为 Object不是一个函数式接口。

JavaAPI在java.util.function包中定义了许多非常通用的函数式接口（我们将在第2章和第3章中对这些接口进行详细讲解）。其中接口BiFunction<T,U,R＞描述了T和U类型的方法参数及返回类型R。你可以将我们的字符串比较lambda表达式保存在一个该类型的变量中。

|  |
| --- |
| BiFunction<String,String,Integer> comp = (first,second)->Integer.*compare*(first.length(),second.length()); |

但是，这对排序并不能起到什么帮助作用。不存在接收BiFunction作为参数的Arrays.sort方法。如果你之前使用过其他函数式编程语言，你可能会对此感到奇怪。但是对于Java开发人员来说，这再自然不过了。像Comparator这样的接口有着特定的目的，而不仅仅是一个接收参数和返回类型的方法。Java8保留了这一习惯。当你希望使用lambda表达式时，你仍然要牢记表达式的目的，并为它指定一个函数式接口。

现在Java8本身的API使用了java.util.function中的接口，将来这些接口很可能被应用在各个地方。但是请记住，任何一个lambda表达式都可以等价转换成现在所使用的API中对应的函数式接口。

* 注意：

你可以在任意函数式接口上标注＠Functionalinterface注解，这样做有两个好处。

1. 首先，编译器会检查标注该注解的实体，检查它是否是只包含一个抽象方法的接口。
2. 另外，在javadoc页面也会包含一条声明，说明这个接口是一个函数式接口。

该注解并不要求强制使用。从概念上来讲，所有只含有一个抽象方法的接口都是函数式接口，但是使用＠FunctionalInterface注解会让你的代码看上去更清楚。

最后，当一个lambda表达式被转换为一个函数式接口的实例时，请注意处理检查期异常。如果lambda表达式中可能会抛出一个检查期异常，那么该异常需要在目标接口的抽象方法中进行声明。例如，以下表达式会产生一个错误：

|  |
| --- |
| Runnable sleeper = () -> {  System.***out***.println(**"ZzZ"**);  Thread.*sleep*(1000); };//错误：Thread.sleep可以抛出一个检查期的InterruptedExceptiono |

由于Runnable.run不能抛出任何异常，所以这个赋值是不合法的，有两种方法可以修正该问题。一种是在lambda表达式中捕获异常，另一种是将lambda表达式赋给一个其抽象方法可以抛出异常的接口。例如，Callable接口的call方法可以抛出任何异常，因此，你可以将该lambda表达式赋给Callable<Void>（如果你添加一条“return null”语句）。

## 1.4 方法引用

有些时候，你想要传递给其他代码的操作己经有实现的方法了。例如，假设你只想要在按钮被点击时打印event对象，你可以像如下代码一样来调用：

|  |
| --- |
| button.setOnAction(event -> System.***out***.println(event)); |

如果你能够只将println方法传递给setOnAction方法，就更好了。下面是修改后的代码：

|  |
| --- |
| button.setOnAction(System.***out***::println); |

表达式”System.***out***::println”是一个方法引用，等同于lambda表达式“X -> System.out.println(x）”。

我们再举另外一个例子，假设你希望不区分大小写地对字符串进行排序，那么可以传入下面这个方法引用：“Arrays.sort(strings, String::compareToignoreCase)” 正如示例代码所示，**：：**操作符将方法名和对象或类的名字分隔开来。

以下是三种主要的使用情况：

1. 对象：：实例方法
2. 类：：静态方法
3. 类：：实例方法

在前两种情况中，方法引用等同于提供方法参数的lambda表达式。如之前所述：

System.out::println等同于System.out.println(x）。相似的：

Math::pow等同于“（x, y)->Math.pow(x, y）”。

在第三种情况中，第一个参数会成为执行方法的对象。

例如String::cornpareToignoreCase等同于”（x, y)->x.cornpareToignoreCase(y)”

* 注意：

如果有多个同名的重载方法，编译器会试图从上下文中找到最匹配的一个方法。例如，有两个版本的Math.max方法，一个接收整型作为参数，而另个接收double类型的值。究竟会选择哪一个方法，取决于Math:rnax被转换为的函数式接口的方法参数。同lambda表达式类似，方法引用也不会独立存在，它们经常被用于转换为函数式接口的实例。

你还可以捕获方法引用中的this参数。

例如，**this::equals**就等同于**X-> this.equals(x）**。

你也可以使用super对象，以下方法表达式super：：实例方法

会使用this作为执行方法的对象，并调用父类中指定的方法。以下是一些用来说明该机制的虚构示例：

|  |
| --- |
| **class** Greeter {  **public void** greet() {  System.***out***.println(**"Hello worrld"**);  } } **class** ConcurrentGreeter **extends** Greeter {  **public void** greet() {  Thread t1 = **new** Thread(**super**::greet);  t1.start();  } } |

当线程启动时，会调用它的Runnable方法，然后执行super::greet并调用父类中的greet方法。

* **注意：**

**在匿名类中，你可以调用一个闭合类或父类的员工方法，例如闭合类.this：：方法或者闭合类.super：：方法。**

## 1.5 构造器引用

构造器引用同方法引用类似，不同的是在构造器引用中方法名new。

例如，Button::new表示Button类的构造器引用。

对于拥有多个构造器的类，选择使用哪个构造器取决于上下文。假设你有一个字符串列表，并且希望调用Button类的构造器，使用列表中的字符串来构造一个按钮列表，可以使用如下表达式：

|  |
| --- |
| List<String> labels = …; Stream<Button> stream = labels.stream.map(**Button::new**); List<Button> buttons = stream.collection(Collections.toList()); |

我们将会在第2章详细讨论stream、map和collect方法。但是现在，重要的是对于每个列表元素，map方法会调用Button(String）构造器。虽然Button类有多个构造器，但是编译器会从上下文中推断并挑选只带有一个String参数的构造器。

你可以使用数组类型来编写构造器引用。例如，int[ ]::new是一个含有一个参数的构造器引用，这个参数就是数组的长度。它等同于lambda表达式x->new int[x]。

数组构造器引用可以用来绕过Java中的一个限制。在Java中，无法构造一个泛型类型T的数组。表达式new T [ n ]是错误的，因为它会被擦除为new Object[ n ]。这对于编写APl的开发人员来说是一个问题。例如，假设我们希望构造一组按钮，Stream接口中有一个返回Object数组的toArray方法：

Object[ ] buttons=stream.toArray();

但是这并不能让我们满意。用户希望一组按钮对象，而不是一组Object对象。streamAPl通过构造器引用解决了这个问题。它允许将Button[ ]:new传递给toArray方法：

Button[ ] buttons = stream.toArray(Button[ ]::new);toArray方法会调用该构造器来获得一个正确类型的数组。然后它会填充并返回该数组。

## 1.6 变量作用域

通常，我们希望能够在lamba表达式的闭合方法或类中访问其他的变量，例如：

|  |
| --- |
| **public static void** repeatMessage(String text, **int** count) {  Runnable r = () -> {  **for** (**int** i = 0; i < count; i++) {  System.***out***.println(text);  Thread.*yield*();  }  };  **new** Thread(r).start(); } |

假设有以下调用：repeatMessage(“Hello”，1000); //在另一个线程中打印Hello一千次。

注意看lambda表达式中的变量count和text，它们并没有在lambda表达式中被定义，而是方法repeatMessage的参数变量。

如果你思考一下，就会发现这里有一些隐含的东西。lambda表达式可能会在repeatMessage返回之后才运行，此时参数变量己经消失了。如果保留text和count变量会怎样呢？

为了理解这一点，我们需要对lambda表达式有更深入的理解。一个lambda表达式包括三个部分：

* 1：一段代码。
* 2：参数。
* 3：自由变量的值，这里的“自由”指的是那些不是参数并且没有在代码中定义的变量。

在我们的示例中，lambda表达式有两个自由变量，text和count。数据结构表示lambda表达式必须存储这两个变量的值，即”Hello”和1000。我们可以说，这些值己经被lambda表达式捕获了。（这是一个技术实现的细节。例如，你可以将一个lambda表达式转换为一个只含一个方法的对象，这样自由变量的值就会被复制到该对象的实例变量中。）

注意：含有自由变量的代码块被称之为“闭包（closure）”。如果有人得意洋洋地宣传他们的语言有闭包，你可以放心，Java也有。在Java申，lambda表达式就是闭包。事实上，内部类直都是闭包。Java8中为闭包赋予了更吸引人的语法。

System.out.println(text);Thread.yield();

newThread(r).start();

假设有以下调用：

repeatMessage("Hello”，1000);／／在另一个线程中打印Hello一千次

注意看lambda表达式中的变量count和text，它们并没有在lambda表达式中被定义，而是方法repeatMessage的参数变量。

如果你思考一下，就会发现这里有一些隐含的东西。lambda表达式可能会在repeat

Message返回之后才运行，此时参数变量己经消失了。如果保留text和count变量会怎样呢？

为了理解这一点，我们需要对lambda表达式有更深入的理解。一个lambda表达式包括三个部分：

I. 一段代码。

2. 参数。

3. 自由变量的值，这里的“自由”指的是那些不是参数并且没有在代码中定义的变量。

在我们的示例中，lambda表达式有两个自由变量，text和count。数据结构表示lambda表达式必须存储这两个变量的值，即”Hello”和1000。我们可以说，这些值己经被lambda表达式捕获了。（这是一个技术实现的细节。例如，你可以将一个lambda表达式转换为一个只含一个方法的对象，这样自由变量的值就会被复制到该对象的实例变量中。〉

注意：含有自由变量的代码块被称之为“闭包（closure）”。如果有人得意洋洋地宣传他们的语言有闭包，你可以放心，Java也有。在Java申，lambda表达式就是闭包。事实上，内部类直都是闭包。Java8中为闭包赋予了更吸引人的语法。

如你所见，lambda表达式可以捕较闭合作用域中的变量值。在Java中，为了确保被

＜王1

捕获的值是被良好定义的，需要遵守一个重要的约束。在lambda表达式中，被引用的变量的值不可以被更改。例如，下面这个表达式是不合法的：

口D publicstaticvoidrepeatMessage(Stringtext,intcount){

Runnabler=()->{while(count>0){

count--;II牵制足，不能更改已捕获变莹的位

System.out.println(text);Thread.yield();

newThread(r).start();

做出这个约束是有原因的。更改lambda表达式中的变量不是线程安全的。假设有一系列并发的任务，每个线程都会更新一个共享的计数器。

intmatches=0;

for(Pathp:files)

newThread(（）→｛if(p中包含某些属性）matches++;)).start();

／／非法更改matches的位

如果这段代码是合法的，那么会引起十分糟糕的结果。自增操作matches忡不是原子操作，如果多个线程并发执行该自增操作，天晓得会发生什么。

注意：内部类也会捕获闭合作用域中的值。在Java8之前，内部类只允许访问final的局部变量。为了适应lambda表达式，这条规则现在也被放宽了。一个内部类可以访问任何有效的final局部变量一巨P任何值不会发生变化的变量。

不要指望编译器会捕获所有并发访问错误。不可变的约束只作用在局部变量上。如果matches是一个实例变量或者某个闭合类的静态变量，那么不会有任何错误被报告出来，即使结果同样未定义。

同样，改变一个共享对象也是完全合法的，即使这样并不恰当。例如，

List<Path>matches=newArrayList<>();for(Pathp:files)

newThread(()->(if(p中包含某些属性）matches.add(p);}).start();

／／你可以改变matches的位．但是在多线程下是不安全的

注意matches是“有效final”的（一个有效的final变量被初始化后，就永远不会再被赋予一个新值的变量〉。在我们的示例中，matches总是引用同一个ArrayList ＜五1对象。但是，这个对象是可变的，因此是线程不安全的。如果多个线程同时调用add万

法，结果将无法预测。

其实也有能够并发计数和收集值的线程安全的机制。在第2章中，你将会学习如何使用stream来收集具有某些属性的值。在其他情况下，你可能希望使用线程安全的计数器和集合。关于这一重要话题的更多信息，请参考第6章。

注意：同内部类 样，我们也有 种巧妙的方式，能够让lambda表达式更新一个闭合、局部作用域中的计数器。我们可以使用一个长度为l的数组，如下所示。

int[]counter=new工nt[l];

button.setOnAction(event->counter[OJ++);

当然，像这样的代码不是线程安全的。对于一个按钮的回调方法来说，这无关紧要，但是一般来说，如果你真的想使用这种小技巧，还是应当三思而行。在第6章你将会学习如何实现一个线程安全的共享计数器。

lambda表达式的方法体与嵌套代码块有着相同的作用域。因此它也适用同样的命名冲突和屏蔽规则。在lambda表达式中不允许声明一个与局部变量同名的参数或者局部变量。

Pathfirst=Paths.get（”／usr/bin”）;

Comparator<String>comp=

(first,second)->Integer.compare(first.length(),second.length());

／／错误：变量first已经被定义了

在一个方法中，你不能有两个同名的局部变量，因此，你也不能在lambda表达式中引入这样的变量。

当你在lambda表达式中使用this关键字时，你会引用创建该lambda表达式的方法的this参数，以下面的代码为例：

publicclassApplication{publicvoiddoWork(){

Runnablerunner=()>{...;System.out.pr工ntln(this.toString());...};

表达式this.toString（）会调用Application对象的toString方法，而不是Runnable实例的toString方法。在lambda表达式中使用this，与在其他地方使用this没有什么不同。lambda表达式的作用域被嵌套在doWork方法中，并且无论this位于方法的何处，其意义都是一样的。

口王＞ 1.7 默认为法

许多开发语言都将函数表达式集成到了其集合库中。这样比循环方式所需的代码更少，并且更加容易理解。以下面的循环为例：

for(inti=O;i<list.size();i++)System.out.pr工ntln(list.get(i));

事实上有一种更好的方式。API开发人员可以提供一个forEach方法，用来将一个函数应用到集合的每个元素上。下面是使用这种方式编写的一个简单调用：

list.forEach(System.out::println);

如果集合库是完全重新设计的，这样做不会有什么问题。但是，Java的集合库是许多年以前设计的，这就会带来一个问题。如果Collection接口添加了新的方法，例如forEach，那么每个实现了Collection接口的自定义类就必须都实现该方法。这在Java中是完全无法接受的。

Java设计者们希望通过允许接口包含带有具体实现的方法（称为默认方法）来一劳永逸地解决这个问题。这些方法可以被安全地添加到己有的接口中。在本节中，我们将详细讲解默认方法。

注意：在Java8中，通过将要在本节中介绍的机制，forEach方法已经被添加歪IJIterable撞口中（它是Collect工on接口的 个父接口）。

假设有如下接口：interfacePerson(longgetld();

defaultStringgetName()(return”JohnQ.Public";)

该接口有两个方法：一个抽象方法getId，以及一个默认方法getName。当然，实现Person接口的具体类必须实现getId方法，但是它可以选择保留getName的实现，或者重写它。

默认方法终结了以前的一种经典模式，即提供一个接口，以及一个实现接口的大多数或全部方法的抽象类，例如Collection/AbstractCollection或WindowListener/WindowAdapter。现在你只需要在接口中实现那些方法。

如果一个接口中定义了一个默认方法，而另外一个父类或接口中又定义了一个同名 ＜至1

的方法，该选择哪个呢？像Scala和C＋＋等语言可能会有一套复杂的规则来解决这种二义性，但是幸运的是，Java中的规则要简单得多，如下所示：

I. 选择父类中的方法。如果一个父类提供了具体的实现方法，那么接口中具有相同名称和参数的默认方法会被忽略。

2.接口冲突。如果一个父接口提供了一个默认方法，而另一个接口也提供了一个具有相同名称和参数类型的方法（不管该方法是否是默认方法），那么你必须通过覆盖该方法来解决冲突。

我们来详细理解一下第二条规则。假定另一个接口也含有一个名为getName的方法：

interfaceNamed{

defaultStringgetName(){returngetClass().getName()＋””＋hashCode();}

如果你编写了一个同时实现这两个接口的类，会发生什么事呢？

classStudentimplementsPerson,Named{

该类会继承由Person和Named接口同时提供的getName方法，但是这两个方法的实现并不一致。Java编译器会报告一个错误，并交由开发人员来解决这种冲突，而不会自动选择其中一个。对于这种情况，你只需要在Student类中提供一个getName方法，在该方法中再选择调用其中一个接口中的方法，如下所示：

classStudentimplementsPerson,Named{

publicStringgetName(){returnPerson.super.getNa皿阜（）;)

第1章lambda表达式

现在我们假定Named接口没有提供getName方法的一个默认实现：

interfaceNamed{

StringgetName();

如果这样，Student类能继承Person接口中的默认方法吗？也许这说得过去，但是Java设计者们为了保持统一，还是选择了与之前一样的处理方式。两个接口如何冲突并不重要，只要有一个接口提供了实现，编译器就会报告一个错误，而开发人员必须手动解决这种冲突。

OD

回注意：当然，如果两个接口都没有为共享方法提供一个默认的实现，那么我们

就又回到了Java8之前的情况，也就不存在什么冲突。这时实现类有两种选择，

即实现该方法或者不实现该方法。如果选择后者，该类本身就是 个捕象类。

我们刚刚讨论了两个接口之间的冲突。现在我们考虑这样一个类，它继承了某个父类并实现了某个接口，而这个父类和接口中都有一个同名的方法。例如，假设Person是一个类，而Student类的定义如下所示：

classStudentextendsPersonimplementsNamed{...}

在这种情况下，只有父类中的方法会起作用，接口中的任何默认方法都会被忽略。在这个例子中，不管Named接口中的getName方法是否是默认方法，Student都会继承Person类中的getName方法。这就是“类优先”的规则。

“类优先”规则可以保证与Java7的兼容性。如果你在接口中添加了一个默认方法，它对Java8以前编写的代码不会产生任何影响。

囹小心：你不能为Object中的方法重新定义一个默认方法。例如，你不能定义

－个默认方法toString或者equals，即使这对于List这样的接口很有吸引

力。“类优先”规则的结果是，这样的方法永远不可能优先于

Object.toString或者Object.equals。

1.8 接口中的静态方法

在Java8中，你可以为接口添加静态方法。从技术角度来说，这是完全合法的。只是它看起来违反了接口作为一个抽象定义的理念。

至今为止，我们经常在相互一起使用的类中使用静态方法。你可以在标准库中找到像Collection/Collections或者Path/Paths这样成对的接口和类。

以Paths类为例，它只有一些工厂方法。你可以通过一组字符串来构造一个路径，例如Paths.get（”jdkl.8.”，”jre”，”bin”）。在Java8中，Path接口中添加了这样一个方法：

publicinterfacePath{

publicstaticPathget(Stringfirst,String...more){

returnFileSystems.getDefault().getPath(first,more);

＜互3

这样Paths类就再没有存在的必要了。

当你查看Collections类时，你会发现两种方法，一种如下所示：

publicstaticvoidshuffle(List<?>list)

这类方法可以作为List接口中的一个默认方法：

publicdefaultvoidshuffle()

然后你只需要在任何list对象上调用list.shuffle（）。但是这样无法适用于静态的工厂方法，因为你没有可用的对象来调用该方法。因此，

Java8中引入了静态接口方法。例如：

publicstatic<T>List<T>nCopies(intn,To)

／／构造含有n个o类型实例的列表

它可以作为List接口的一个静态方法。这样你就可以调用List.nCopies(1O,"Fred”），而不是Collections.nCopies(lO，”Fred”），就能够更清楚地表示返回的结果是一个List对象。

虽然Java的集合类不太可能像这样进行重构，但是当你实现自己的库时，不必再为工具方法提供一个辅助的类了。

在Java8中，很多接口中已经添加了静态方法。例如，Comparator接口提供了一

个很实用的比较方法，它接收一个“键提取”函数，并产生一个用来比较所提取出的键的比较器。例如，要根据名称对Person对象进行比较，你可以使用函数Comparator.comparing(Person::getname）。

在本章中，我们曾经使用lambda表达式（first,second)->Integer.compare(first.length(),second.length（））对字符串的长度进行比较。但是如果使用静态的比较方法，代码会变得更加简洁，例如Comparator.comparing(String::length）。这个例子很适合作为本章的结尾，因为它充分展示了函数的强大功能。compare方法将一个函数（键提取〉转变为了一个更复杂的函数（基于键的比较器〉。我们将会在第3章中更详细地介绍这种“高阶函数”。

口D 练习

1.Array.sort方法中的比较器代码的线程与调用sort的线程是同一个吗？

2. 使用java.io.File类的listF工les(FileFilter）和isDirectory方法，编写一个返回指定目录下所有子目录的方法。使用lambda表达式来代替FileFilter对象，再将它改写为一个方法引用。

3. 使用java.工o.File类的list(FilenameFilter）方法，编写一个返回指定目录下、具有指定扩展名的所有文件。使用lambda表达式（而不是FilenameFilter)来实现。它会捕获闭合作用域中的哪些变量？

4.对于一个指定的File对象数组，首先按照路径的目录排序，然后对每组目录中的元素再按照路径名排序。便请用lambda表达式（而不是Comparator）来实现。

5.从你的项目中选取个包含一些ActionListener、Runnable或者其他类似代码的文件。将它们替换为lambda表达式。这样能节省多少行代码？替换后的代码是否具有更好的可读性？在这个过程中你使用了方法引用吗？

6.你是否讨厌在Runnable实现中处理检查器异常？编写一个捕获所有异常的

uncheck方法，再将它改造为不需要检查异常的方法。例如：

newThread(uncheck(

(}->{System.out.println（”ZzZ”）；Thread.sleep(lOOO);))).start(};

／／看，不需妥catch(InterruptedException)!

提示：定义一个RunnableEx接口，其run方法可以抛出任何异常。然后实现

publicstaticRunnableuncheck(RunnableExrunner）。在uncheck

函数中使用一个lambda表达式。

为什么你不能直接使用Callable<Void＞宋代替RunnableEx?

7.编写一个静态方法andThen，它接收两个Runnable实例作为参数，并返回一个分别运行这两个实例的Runnable对象。在main方法中，向andThen方法传递两个lamba表达式，并运行返回的实例。

8.当一个lambda表达式捕获了如下增强for循环中的值时，会发生什么？

String[]names=｛”Peter”，”Paul”，”Mary"};List<Runnable>runners=newArrayList<>();

for(Stringname:names)

runners.add(()->System.out.println(name));

这样做是否合法？每个lambda表达式都捕获了一个不同的值，还是它们都获得了最 〈豆1

终的值？如果使用传统的for循环，例如for(inti=O;i<names.length;

i＋＋），又会发生什么？

9.编写一个继承Collection接口的子接口Collect工on2，并添加一个默认方法voidforEachif(Consumer<T>action,Predicate<T>filter），用来将action应用到所有filter返回true的元素上。你能够如何使用它？

10.浏览Collections类中的方法。如果哪一天你可以做主，你会将每个方法放到哪个接口中？这个方法会是一个默认方法还是静态方法？

门．假设你有一个实现了两个接口I和J的类，这两个接口都有一个voidf（）方法。如果I接口中的f方法是一个抽象的、默认或者静态方法，并且J接口中的f方法也是一个抽象的、默认或者静态方法，分别会发生什么？如果这个类继承自s类井实现了接口工，并且s和工中都有一个voidf（）方法，又分别会发生什么？

12.在过去，你知道向接口中添加方法是一种不好的形式，因为它会破坏己有的代码。现在你知道了可以向接口中添加新方法，同时能够提供一个默认的实现。这样做安全程度如何？描述一个Collection接口的新stream方法会导致遗留代码编译失败的场景。二进制的兼容性如何？JAR文件中的遗留代码是否还能运行？