

- 1 lambda表达式的语法
- 2 函数式接口
- 3 方法引用
- 4 构造器引用
- 5 变量作用域
- 6 处理lambda表达式
- 7 Comparator接口

lambda表达式是一个可传递的代码块，可以在以后执行一次或多次。

## 1 lambda表达式的语法

lambda表达式的语法：参数+箭头+表达式。参数就像方法参数一样，指定参数类型和名称，放在小括号中，多个参数之间用逗号分隔。例如：

```
(String first, String second) -> first.length() - second.length()
```

无须指定lambda表达式的返回类型，其返回类型由上下文推导得出。

如果代码要完成的计算无法放在一个表达式中，就要把这些代码放在大括号中，并包含显式的 `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;
}
```

如果一个lambda表达式只在某些分支返回一个值，而另一些分支不返回值，这是不合法的。例如：

```
(int x) -> { if (x >= 0) return 1; } // 不合法
```

即使lambda表达式没有参数，仍然要提供空括号。例如：

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

如果可以推导出lambda表达式的参数类型，就可以省略其类型。例如：

```
Comparator<String> comp
    = (first, second) -> first.length() - second.length();
```

如果只有一个参数，而且这个参数的类型可以推导得出，就可以省略小括号和类型。例如：

```
ActionListener listener = event ->
    System.out.println("The time is " + Instant.ofEpochMilli(event.getWhen()));
```

## 2 函数式接口

对于只有一个抽象方法的接口，需要这种接口的对象时，可以提供一个lambda表达式，用于实现抽象方法。这种接口称为**函数式接口**。例如，`Comparator` 接口就是只有一个抽象方法的接口，可以向它提供一个lambda表达式：

```
Arrays.sort(words,
    (first, second) -> first.length() - second.length());
```

`Arrays.sort` 方法会接收实现了 `Comparator<String>` 的某个类的对象，在这个对象上调用 `compare` 方法会执行这个lambda表达式的体。

Java API在 `java.util.function` 包中定义了很多非常通用的函数式接口。其中 `Predicate` 接口的定义如下：

```
public interface Predicate<T>
{
    boolean test(T t);
    // 其他默认和静态方法
}
```

`ArrayList` 类有一个 `removeIf` 方法，它的签名为：

```
boolean removeIf(Predicate<? super E> filter)
    // 删除列表中满足条件的所有元素。条件由 filter 指定
    // 如果有元素被删除，返回 true，否则返回 false
```

使用 `removeIf` 方法时，可以向参数传递一个lambda表达式作为筛选条件，例如：

```
list.removeIf(e -> e == null); // 删除列表中所有 null 值
```

## 3 方法引用

方法引用可以用来代替lambda表达式。方法引用的语法主要有3种情况：

1. `object::instanceMethod`。前面是对象，后面是实例方法名，中间用双冒号分隔。等价于向方法传递参数的lambda表达式。例如 `System.out::println`，`System.out` 是对象，`println` 是方法名，它等价于 `x -> System.out.println(x)`。
2. `Class::instanceMethod`。前面是类名，后面是实例方法名，中间用双冒号分隔。等价于lambda表达式的第一个参数作为隐式参数，其他参数作为显式参数，传递给方法。例如 `String::compareToIgnoreCase` 等价于 `(x, y) -> x.compareToIgnoreCase(y)`。
3. `Class::staticMethod`。前面是类名，后面是静态方法名，中间用双冒号分隔。等价于lambda表达式的所有参数都传递到静态方法。例如 `Math::pow` 等价于 `(x, y) -> Math.pow(x, y)`。

下表提供了更多示例：

方法引用	等价的lambda表达式	说明
<code>separator::equals</code>	<code>x -&gt; separator.equals(x)</code>	第一种情况
<code>String::trim</code>	<code>x -&gt; x.trim()</code>	第二种情况
<code>String::concat</code>	<code>(x, y) -&gt; x.concat(y)</code>	第二种情况
<code>Integer::valueOf</code>	<code>x -&gt; Integer.valueOf(x)</code>	第三种情况
<code>Integer::sum</code>	<code>(x, y) -&gt; Integer.sum(x, y)</code>	第三种情况

方法引用指示编译器生成一个函数式接口的实例，覆盖这个接口的抽象方法来调用给定的方法。类似于lambda表达式，方法引用本身不是对象，但为函数式接口变量赋值时会生成一个对象。例如，下面两条语句是等价的：

```
Timer timer = new Timer(1000, event -> System.out.println(event));
Timer timer = new Timer(1000, System.out::println);
```

只有lambda表达式的体只调用一个方法而不做其他操作时，才能把lambda表达式重写为方法引用。例如下面的lambda表达式不能重写为方法引用：

```
s -> s.length() == 0; // 除了方法调用，还有一个比较
```

如果有多个同名的重载方法，编译器就会尝试从上下文中找出你指的是哪一个方法。例如，`Math.max`方法有两个版本，一个用于整数，另一个用于`double`值。选择哪个版本取决于`Math::max`转换为哪个函数式接口的方法参数。类似于lambda表达式，方法引用不能独立存在，总是会转换为函数式接口的实例。

包含对象的方法引用与等价的lambda表达式还有一个细微的差别。例如`separator::equals`，如果`separator`为`null`，构造`separator::equals`时就会立即抛出一个`NullPointerException`异常，而lambda表达式`x -> separator.equals(x)`只在调用时才会抛出`NullPointerException`。

可以在方法引用中使用`this`参数。例如，`this::equals`等价于`x -> this.equals(x)`，属于第一种情况。

使用`super`也是合法的，`super::instanceMethod`使用`this`作为隐式参数，调用给定方法的超类版本。

## 4 构造器引用

构造器引用与方法引用很类似，语法是`Class::new`，前面是类名，后面用`new`作为方法名，中间用双冒号分隔。如果有多个重载的构造器，将根据上下文决定使用哪个构造器。

可以用数组类型建立构造器引用。例如，`int[]::new`是一个构造器引用，它有一个参数，用于指定数组长度。这个构造器引用等价于lambda表达式`x -> new int[x]`。

## 5 变量作用域

lambda表达式有3个部分：代码块、参数、自由变量。这里的自由变量是指非参数并且不在代码块中定义的变量。例如：

```
public static void repeatMessage(String text, int delay)
{
    ActionListener listener = event ->
    {
        System.out.println(text); // text 是自由变量
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

lambda表达式中的变量 `text` 既不是参数，也不是在lambda表达式的代码块中定义的，它是外围方法的一个参数变量，因此 `text` 就是自由变量。表示lambda表达式的数据结构必须存储自由变量的值，称自由变量的值被lambda表达式捕获。

lambda表达式可以捕获外围作用域中变量的值，要确保所捕获的值是明确定义的。在lambda表达式中，只能引用不会改变的变量，不能在lambda表达式内部修改自由变量。例如，下面的做法是不合法的：

```
public static void countDown(int start, int delay)
{
    ActionListener listener = event ->
    {
        start--; // 错误
        System.out.println(start);
    };
    new Timer(delay, listener).start();
}
```

如果在lambda表达式中引用一个变量，而这个变量可能在外部改变，这也是不合法的。例如：

```
public static void repeat(String text, int count)
{
    for (int i = 1; i <= count; i++)
    {
        ActionListener listener = event -> System.out.println(i + ": " + text);
        // 错误
        new Timer(1000, listener).start();
    }
}
```

lambda表达式中捕获的变量必须是事实最终变量，即这个变量初始化之后就不会再为它赋新值。在上面的例子中，`text` 和 `i` 都是自由变量。由于字符串是不可变的，而且 `text` 总是引用同一个 `String` 对象，所以捕获 `text` 是合法的。不过，`i` 的值会改变，因此不能捕获 `i`。

lambda表达式的体与嵌套块有相同的作用域。这里同样适用命名冲突和遮蔽的有关规则。在lambda表达式中声明一个与局部变量同名的参数或局部变量是不合法的。

在一个方法中不能有两个同名的局部变量，lambda表达式中同样也不能有同名的局部变量。

在一个lambda表达式中使用 `this` 关键字时，是指创建这个lambda表达式的方法的 `this` 参数。例如：

```

public class Application
{
    public void init()
    {
        ActionListener listener = event ->
        {
            System.out.println(this.toString());
            //...
        }
        //...
    }
}

```

表达式 `this.toString` 会调用 `Application` 对象的 `toString` 方法。在 `lambda` 表达式中，`this` 的使用并没有任何特殊之处。`lambda` 表达式的作用域嵌套在 `init` 方法中，与出现在这个方法中的其他位置一样，`lambda` 表达式中 `this` 的含义并没有变化。

## 6 处理lambda表达式

下表列出了Java API提供的最重要的函数式接口：

函数式接口	参数类型	返回类型	抽象方法名	描述	其他方法
<code>Runnable</code>	无	<code>void</code>	<code>run</code>	作为无参数或返回值的动作运行	

函数式接口	参数类型	返回类型	抽象方法名	描述	其他方法
<code>Supplier&lt;T&gt;</code>	无	T	<code>get</code>	提供一个 T 类型的值	
<code>Consumer&lt;T&gt;</code>	T	<code>void</code>	<code>accept</code>	处理一个 T 类型的值	<code>andThen</code>
<code>BiConsumer&lt;T, U&gt;</code>	T, U	<code>void</code>	<code>accept</code>	处理 T 和 U 类型的值	<code>andThen</code>
<code>Function&lt;T, R&gt;</code>	T	R	<code>apply</code>	有一个 T 类型参数的函数	<code>compose</code> , <code>andThen</code> , <code>identity</code>

函数式接口	参数类型	返回类型	抽象方法名	描述	其他方法
<code>BiFunction&lt;T, U, R&gt;</code>	T, U	R	<code>apply</code>	有 T 和 U 类型参数的函数	<code>andThen</code>
<code>UnaryOperator&lt;T&gt;</code>	T	T	<code>apply</code>	类型 T 上的一元操作符	<code>compose</code> , <code>andThen</code> , <code>identity</code>
<code>BinaryOperator&lt;T&gt;</code>	T, T	T	<code>apply</code>	类型 T 上的二元操作符	<code>andThen</code> , <code>maxBy</code> , <code>minBy</code>
<code>Predicate&lt;T&gt;</code>	T	<code>boolean</code>	<code>test</code>	布尔值函数	<code>and</code> , <code>or</code> , <code>negate</code> , <code>isEqual</code>

函数式接口	参数类型	返回类型	抽象方法名	描述	其他方法
<code>BiPredicate&lt;T, U&gt;</code>	<code>T, U</code>	<code>boolean</code>	<code>test</code>	有两个参数的布尔值函数	<code>and</code> , <code>or</code> , <code>negate</code>

下表列出了基本类型 `int`、`long` 和 `double` 的34个可用的特殊化接口：



函数式接口	参数类型	返回类型	抽象方法名
<code>BooleanSupplier</code>	无	<code>boolean</code>	<code>getAsBoolean</code>
<code>IntSupplier</code>	无	<code>int</code>	<code>getAsInt</code>
<code>LongSupplier</code>	无	<code>long</code>	<code>getAsLong</code>
<code>DoubleSupplier</code>	无	<code>double</code>	<code>getAsDouble</code>
<code>IntConsumer</code>	<code>int</code>	<code>void</code>	<code>accept</code>
<code>LongConsumer</code>	<code>long</code>	<code>void</code>	<code>accept</code>
<code>DoubleConsumer</code>	<code>double</code>	<code>void</code>	<code>accept</code>
<code>ObjIntConsumer&lt;T&gt;</code>	<code>T, int</code>	<code>void</code>	<code>accept</code>
<code>ObjLongConsumer&lt;T&gt;</code>	<code>T, long</code>	<code>void</code>	<code>accept</code>
<code>ObjDoubleConsumer&lt;T&gt;</code>	<code>T, double</code>	<code>void</code>	<code>accept</code>
<code>IntFunction&lt;T&gt;</code>	<code>int</code>	<code>T</code>	<code>apply</code>
<code>LongFunction&lt;T&gt;</code>	<code>long</code>	<code>T</code>	<code>apply</code>
<code>DoubleFunction&lt;T&gt;</code>	<code>double</code>	<code>T</code>	<code>apply</code>
<code>IntToLongFunction</code>	<code>int</code>	<code>long</code>	<code>applyAsLong</code>
<code>IntToDoubleFunction</code>	<code>int</code>	<code>double</code>	<code>applyAsDouble</code>
<code>LongToIntFunction</code>	<code>long</code>	<code>int</code>	<code>applyAsInt</code>
<code>LongToDoubleFunction</code>	<code>long</code>	<code>double</code>	<code>applyAsDouble</code>
<code>DoubleToIntFunction</code>	<code>double</code>	<code>int</code>	<code>applyAsInt</code>
<code>DoubleToLongFunction</code>	<code>double</code>	<code>long</code>	<code>applyAsLong</code>
<code>ToIntFunction&lt;T&gt;</code>	<code>T</code>	<code>int</code>	<code>applyAsInt</code>
<code>ToLongFunction&lt;T&gt;</code>	<code>T</code>	<code>long</code>	<code>applyAsLong</code>
<code>ToDoubleFunction&lt;T&gt;</code>	<code>T</code>	<code>double</code>	<code>applyAsDouble</code>
<code>ToIntBiFunction&lt;T, U&gt;</code>	<code>T, U</code>	<code>int</code>	<code>applyAsInt</code>
<code>ToLongBiFunction&lt;T, U&gt;</code>	<code>T, U</code>	<code>long</code>	<code>applyAsLong</code>
<code>ToDoubleBiFunction&lt;T, U&gt;</code>	<code>T, U</code>	<code>double</code>	<code>applyAsDouble</code>
<code>IntUnaryOperator</code>	<code>int</code>	<code>int</code>	<code>applyAsInt</code>
<code>LongUnaryOperator</code>	<code>long</code>	<code>long</code>	<code>applyAsLong</code>
<code>DoubleUnaryOperator</code>	<code>double</code>	<code>double</code>	<code>applyAsDouble</code>
<code>IntBinaryOperator</code>	<code>int, int</code>	<code>int</code>	<code>applyAsInt</code>

函数式接口	参数类型	返回类型	抽象方法名
<code>LongBinaryOperator</code>	<code>long , long</code>	<code>long</code>	<code>applyAsLong</code>
<code>DoubleBinaryOperator</code>	<code>double , double</code>	<code>double</code>	<code>applyAsDouble</code>
<code>IntPredicate</code>	<code>int</code>	<code>boolean</code>	<code>test</code>
<code>LongPredicate</code>	<code>long</code>	<code>boolean</code>	<code>test</code>
<code>DoublePredicate</code>	<code>double</code>	<code>boolean</code>	<code>test</code>

在自定义方法中使用lambda表达式时，要选择合适的函数式接口。

## 7 Comparator接口

`Comparator` 接口包含很多方便的静态方法来创建比较器，这些方法可以用于lambda表达式或方法引用。

静态 `comparing` 方法取一个“键提取器”函数，它将类型T映射为一个可比较的类型。对要比较的对象应用这个函数，然后对返回的键完成比较。它的签名为：

```
static <T, U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T, ? extends U> keyExtractor)
// T 为要比较的元素的类型，U 为键的类型
// 从 T 类型提取可比较的键，并根据键值进行比较
```

例如，有一个Person对象数组，要按照名字对这些对象进行排序，可以如下实现：

```
Arrays.sort(people, Comparator.comparing(Person::getName));
// 从 Person 类提取名字作为键，按照名字进行排序
```

`comparing` 方法有一个重载版本，允许通过第二个参数指定排序方式。它的签名为：

```
static <T, U> Comparator<T> comparing(Function<? super T, ? extends U> keyExtractor,
                                     Comparator<? super U> keyComparator)
```

例如，要对上面的Person对象数组按照人名长度进行排序，可以如下实现：

```
Arrays.sort(people, Comparator.comparing(Person::getName,
(s, t) -> Integer.compare(s.length(), t.length())));
```

为了避免 `int`、`long` 和 `double` 值的自动装箱，可以使用 `comparingInt`、`comparingLong`、`comparingDouble` 方法。它们只有一个参数，用法与一个参数的 `comparing` 方法类似。例如，要对上面的Person对象数组按照人名长度进行排序，也可以如下实现：

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

`thenComparing` 方法用于指定第二排序键，当第一排序键相同时，按照第二排序键确定顺序。它的用法与 `comparing` 方法类似，也有 `int`、`long` 和 `double` 值的变体。例如：

```
Arrays.sort(people,
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName));
// 先按照 last name 排序, 如果 last name 相同, 再按照 first name 排序
```

可以连续使用多个 `thenComparing` 方法, 指定多个排序键。

如果键值可能为 `null`, 就要用到 `nullsFirst` 和 `nullsLast` 方法。这两个方法会修改现有的比较器, 从而在遇到 `null` 时不会抛出异常。它们的签名为:

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) // null 值
小于正常值, 将 null 值排在前面
static <T> Comparator<T> nullsLast(Comparator<? super T> comparator) // null 值大
于正常值, 将 null 值排在后面
// 如果键值都是 null, 二者相等。如果键值都是正常值, 则按照参数指定的比较器确定顺序
// 如果参数指定的比较器为 null, 则所有正常值都相同
```

例如, 按照人的 middle name 排序, 当一个人没有 middle name 时, `getMiddleName` 会返回 `null`, 此时就可以使用 `nullsFirst` 或 `nullsLast`。这两个方法需要一个比较器作为参数, 指定排序方式。例如, 要按照 middle name 的长度排序, 可以实现如下:

```
Arrays.sort(people, Comparator.comparing(Person::getMiddleName,
Comparator.nullsFirst(Comparator.comparingInt(p -> p.getName().length()))));
```

`naturalOrder` 静态方法可以为任何实现了 `Comparable` 接口的类建立一个比较器, 按照自然顺序进行排序。`reverseOrder` 静态方法会提供自然顺序的逆序。例如:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
Comparator.reverseOrder()));
// 按照姓名进行逆序排序
```

`reversed` 方法将当前比较器转为逆序, 例如:

```
Arrays.sort(people, Comparator.comparing(Person::getName).reversed());
```