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：`(参数列表) -> 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：

```
// 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
boolean removeIf(Predicate<? super E> filter)
```

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
`separator::equals`	`x -> separator.equals(x)`	第一种情况
`String::trim`	`x -> x.trim()`	第二种情况
`String::concat`	`(x, y) -> x.concat(y)`	第二种情况
`Integer::valueOf`	`x -> Integer.valueOf(x)`	第三种情况
`Integer::sum`	`(x, y) -> Integer.sum(x, y)`	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
`Runnable`	无	`void`	`run`	作为无参数或返回值的动作运行	
`Supplier<T>`	无	`T`	`get`	提供一个 `T` 类型的值	
`Consumer<T>`	`T`	`void`	`accept`	处理一个 `T` 类型的值	`andThen`
`BiConsumer<T, U>`	`T, U`	`void`	`accept`	处理 `T` 和 `U` 类型的值	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	有一个 `T` 类型参数的函数	`compose`，`andThen`，`identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	有 `T` 和 `U` 类型参数的函数	`andThen`
`UnaryOperator<T>`	`T`	`T`	`apply`	类型 `T` 上的一元操作符	`compose`，`andThen`，`identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	类型 `T` 上的二元操作符	`andThen`，`maxBy`，`minBy`
`Predicate<T>`	`T`	`boolean`	`test`	布尔值函数	`and`，`or`，`negate`，`isEqual`
`BiPredicate<T, U>`	`T, U`	`boolean`	`test`	有两个参数的布尔值函数	`and`，`or`，`negate`

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
`BooleanSupplier`	无	`boolean`	`getAsBoolean`
`IntSupplier`	无	`int`	`getAsInt`
`LongSupplier`	无	`long`	`getAsLong`

函数式接口	参数类型	返回类型	抽象方法名
`DoubleSupplier`	无	`double`	`getAsDouble`
`IntConsumer`	`int`	`void`	`accept`
`LongConsumer`	`long`	`void`	`accept`
`DoubleConsumer`	`double`	`void`	`accept`
`ObjIntConsumer<T>`	`T, int`	`void`	`accept`
`ObjLongConsumer<T>`	`T, long`	`void`	`accept`
`ObjDoubleConsumer<T>`	`T, double`	`void`	`accept`
`IntFunction<T>`	`int`	`T`	`apply`
`LongFunction<T>`	`long`	`T`	`apply`
`DoubleFunction<T>`	`double`	`T`	`apply`
`IntToLongFunction`	`int`	`long`	`applyAsLong`
`IntToDoubleFunction`	`int`	`double`	`applyAsDouble`
`LongToIntFunction`	`long`	`int`	`applyAsInt`
`LongToDoubleFunction`	`long`	`double`	`applyAsDouble`
`DoubleToIntFunction`	`double`	`int`	`applyAsInt`
`DoubleToLongFunction`	`double`	`long`	`applyAsLong`
`ToIntFunction<T>`	`T`	`int`	`applyAsInt`
`ToLongFunction<T>`	`T`	`long`	`applyAsLong`
`ToDoubleFunction<T>`	`T`	`double`	`applyAsDouble`
`ToIntBiFunction<T, U>`	`T, U`	`int`	`applyAsInt`
`ToLongBiFunction<T, U>`	`T, U`	`long`	`applyAsLong`
`ToDoubleBiFunction<T, U>`	`T, U`	`double`	`applyAsDouble`
`IntUnaryOperator`	`int`	`int`	`applyAsInt`
`LongUnaryOperator`	`long`	`long`	`applyAsLong`
`DoubleUnaryOperator`	`double`	`double`	`applyAsDouble`
`IntBinaryOperator`	`int` , `int`	`int`	`applyAsInt`
`LongBinaryOperator`	`long` , `long`	`long`	`applyAsLong`
`DoubleBinaryOperator`	`double` , `double`	`double`	`applyAsDouble`
`IntPredicate`	`int`	`boolean`	`test`
`LongPredicate`	`long`	`boolean`	`test`
`DoublePredicate`	`double`	`boolean`	`test`

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：

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

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：

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

`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：

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

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：

```
// null 值小于正常值，将 null 值排在前面
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)

// null 值大于正常值，将 null 值排在后面
static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)

// 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：

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

`reversed()`方法将当前比较器转为逆序，例如：

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