Java8 新特性学习

Lambda表达式

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。可以使代码变的更加简洁紧凑。

基本语法

```
(参数列表) -> {代码块}
```

需要注意：

- 参数类型可省略，编译器可以自己推断
- 如果只有一个参数，圆括号可以省略
- 代码块如果只是一行代码，大括号也可以省略
- 如果代码块是一行，且是有结果的表达式，`return`可以省略

注意：事实上，把Lambda表达式可以看做是匿名内部类的一种简写方式。当然，前提是这个匿名内部类对应的必须是接口，而且接口中必须只有一个函数！Lambda表达式就是直接编写函数的：参数列表、代码体、返回值等信息，**用函数来代替完整的匿名内部类**！

用法示例

示例1：多个参数

准备一个集合：

```
// 准备一个集合
List<Integer> list = Arrays.asList(10, 5, 25, -15, 20);
```

假设我们要对集合排序，我们先看JDK7的写法，需要通过匿名内部类来构造一个`Comparator`：

```
// Jdk1.7写法
Collections.sort(list,new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
});
System.out.println(list);// [-15, 5, 10, 20, 25]
```

如果是jdk8，我们可以使用新增的集合API：`sort(Comparator c)`方法，接收一个比较器，我们用Lambda来代替`Comparator`的匿名内部类：

```
// Jdk1.8写法，参数列表的数据类型可省略：
list.sort((i1,i2) -> { return i1 - i2;});

System.out.println(list);// [-15, 5, 10, 20, 25]
```

对比一下`Comparator`中的`compare()`方法，你会发现：这里编写的Lambda表达式，恰恰就是`compare()`方法的简写形式，JDK8会把它编译为匿名内部类。是不是简单多了！

别着急，我们发现这里的代码块只有一行代码，符合前面的省略规则，我们可以简写为：

```
// Jdk8写法
// 因为代码块是一个有返回值的表达式，可以省略大括号以及return
list.sort((i1,i2) -> i1 - i2);
```

示例2：单个参数

还以刚才的集合为例，现在我们想要遍历集合中的元素，并且打印。

先用jdk1.7的方式：

```
// JDK1.7遍历并打印集合
for (Integer i : list) {
    System.out.println(i);
}
```

jdk1.8给集合添加了一个方法：`foreach()`，接收一个对元素进行操作的函数：

```
// JDK1.8遍历并打印集合，因为只有一个参数，所以我们可以省略小括号：
list.forEach(i -> System.out.println(i));
```

实例3：把Lambda赋值给变量

Lambda表达式的实质其实还是匿名内部类，所以我们其实可以把Lambda表达式赋值给某个变量。

```
// 将一个Lambda表达式赋值给某个接口：
Runnable task = () -> {
    // 这里其实是Runnable接口的匿名内部类，我们在编写run方法。
    System.out.println("hello lambda!");
};
new Thread(task).start();
```

不过上面的用法很少见，一般都是直接把Lambda作为参数。

示例4：隐式final

Lambda表达式的实质其实还是匿名内部类，而匿名内部类在访问外部局部变量时，要求变量必须声明为`final`！不过我们在使用Lambda表达式时无需声明`final`，这并不是说违反了匿名内部类的规则，因为Lambda底层会隐式的把变量设置为`final`，在后续的操作中，一定不能修改该变量：

正确示范：

```
// 定义一个局部变量
int num = -1;
Runnable r = () -> {
    // 在Lambda表达式中使用局部变量num，num会被隐式声明为final
    System.out.println(num);
};
new Thread(r).start();// -1
```

错误案例：

```
// 定义一个局部变量
int num = -1;
Runnable r = () -> {
    // 在Lambda表达式中使用局部变量num，num会被隐式声明为final，不能进行任何修改操作
    System.out.println(num++);
};
new Thread(r).start();//报错
```

函数式接口

经过前面的学习，相信大家对于Lambda表达式已经有了初步的了解。总结一下：

- Lambda表达式是接口的匿名内部类的简写形式
- 接口必须满足：内部只有一个函数

其实这样的接口，我们称为函数式接口，我们学过的`Runnable`、`Comparator`都是函数式接口的典型代表。但是在实践中，函数接口是非常脆弱的，只要有人在接口里添加多一个方法，那么这个接口就不是函数接口了，就会导致编译失败。Java 8提供了一个特殊的注解`@FunctionalInterface`来克服上面提到的脆弱性并且显示地表明函数接口。而且jdk8版本中，对很多已经存在的接口都添加了`@FunctionalInterface`注解，例如`Runnable`接口：

```
 * @author  Arthur van Hoff
 * @see     java.lang.Thread
 * @see     java.util.concurrent.Callable
 * @since   JDK1.0
 */
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

另外，Jdk8默认提供了一些函数式接口供我们使用：

Function类型接口

```
@FunctionalInterface
public interface Function<T, R> {
    // 接收一个参数T，返回一个结果R
    R apply(T t);
}
```

Function代表的是有参数，有返回值的函数。还有很多类似的Function接口：

接口名	描述
`BiFunction<T,U,R>`	接收两个T和U类型的参数，并且返回R类型结果的函数
`DoubleFunction<R>`	接收double类型参数，并且返回R类型结果的函数
`IntFunction<R>`	接收int类型参数，并且返回R类型结果的函数
`LongFunction<R>`	接收long类型参数，并且返回R类型结果的函数
`ToDoubleFunction<T>`	接收T类型参数，并且返回double类型结果
`ToIntFunction<T>`	接收T类型参数，并且返回int类型结果
`ToLongFunction<T>`	接收T类型参数，并且返回long类型结果
`DoubleToIntFunction`	接收double类型参数，返回int类型结果
`DoubleToLongFunction`	接收double类型参数，返回long类型结果

看出规律了吗？这些都是一类函数接口，在Function基础上衍生出的，要么明确了参数不确定返回结果，要么明确结果不知道参数类型，要么两者都知道。

Consumer系列

```
@FunctionalInterface
public interface Consumer<T> {
    // 接收T类型参数，不返回结果
    void accept(T t);
}
```

Consumer系列与Function系列一样，有各种衍生接口，这里不一一列出了。不过都具备类似的特征：那就是不返回任何结果。

Predicate系列

```
@FunctionalInterface
public interface Predicate<T> {
    // 接收T类型参数，返回boolean类型结果
    boolean test(T t);
}
```

Predicate系列参数不固定，但是返回的一定是boolean类型。

Supplier系列

```
@FunctionalInterface
public interface Supplier<T> {
    // 无需参数，返回一个T类型结果
    T get();
}
```

Supplier系列，英文翻译就是“供应者”，顾名思义：只产出，不收取。所以不接受任何参数，返回T类型结果。

方法引用

方法引用使得开发者可以将已经存在的方法作为变量来传递使用。方法引用可以和Lambda表达式配合使用。

语法

总共有四类方法引用：

语法	描述
类名::静态方法名	类的静态方法的引用
类名::非静态方法名	类的非静态方法的引用
实例对象::非静态方法名	类的指定实例对象的非静态方法引用
类名::new	类的构造方法引用

示例

首先我们编写一个集合工具类，提供一个方法：

```
    public class CollectionUtil{
        /**
         * 利用function将list集合中的每一个元素转换后形成新的集合返回
         * @param list 要转换的源集合
         * @param function 转换元素的方式
         * @param <T> 源集合的元素类型
         * @param <R> 转换后的元素类型
         * @return
         */
        public static <T,R> List<R> convert(List<T> list, Function<T,R> function){
            List<R> result = new ArrayList<>();
            list.forEach(t -> result.add(function.apply(t)));
            return result;
        }
    }
```

可以看到这个方法接收两个参数：

- `List<T> list`：需要进行转换的集合

- `Function<T,R>`：函数接口，接收T类型，返回R类型。用这个函数接口对list中的元素T进行转换，变为R类型

接下来，我们看具体案例：

类的静态方法引用

```
List<Integer> list = Arrays.asList(1000, 2000, 3000);
```

我们需要把这个集合中的元素转为十六进制保存，需要调用`Integer.toHexString()`方法：

```
public static String toHexString(int i) {
    return toUnsignedString0(i, 4);
}
```

这个方法接收一个i类型，返回一个`String`类型，可以用来构造一个`Function`的函数接口：

我们先按照Lambda原始写法，传入的Lambda表达式会被编译为`Function`接口，接口中通过`Integer.toHexString(i)`对原来集合的元素进行转换：

```
// 通过Lambda表达式实现
List<String> hexList = CollectionUtil.convert(list, i -> Integer.toHexString(i));
System.out.println(hexList);// [3e8, 7d0, bb8]
```

上面的Lambda表达式代码块中，只有对`Integer.toHexString()`方法的引用，没有其它代码，因此我们可以直接把方法作为参数传递，由编译器帮我们处理，这就是静态方法引用：

```
// 类的静态方法引用
List<String> hexList = CollectionUtil.convert(list, Integer::toHexString);
System.out.println(hexList);// [3e8, 7d0, bb8]
```

类的非静态方法引用

接下来，我们把刚刚生成的`String`集合`hexList`中的元素都变成大写，需要借助于String类的toUpperCase()方法：

```
public String toUpperCase() {
    return toUpperCase(Locale.getDefault());
}
```

这次是非静态方法，不能用类名调用，需要用实例对象，因此与刚刚的实现有一些差别，我们接收集合中的每一个字符串`s`。但与上面不同然后`s`不是`toUpperCase()`的参数，而是调用者：

```
// 通过Lambda表达式，接收String数据，调用toUpperCase()
List<String> upperList = CollectionUtil.convert(hexList, s -> s.toUpperCase());
System.out.println(upperList);// [3E8, 7D0, BB8]
```

因为代码体只有对`toUpperCase()`的调用，所以可以把方法作为参数引用传递，依然可以简写：

```
// 类的成员方法
List<String> upperList = CollectionUtil.convert(hexList, String::toUpperCase);
System.out.println(upperList);// [3E8, 7D0, BB8]
```

指定实例的非静态方法引用

下面一个需求是这样的，我们先定义一个数字`Integer num = 2000`，然后用这个数字和集合中的每个数字进行比较，比较的结果放入一个新的集合。比较对象，我们可以用`Integer`的`compareTo`方法:

```
public int compareTo(Integer anotherInteger) {
    return compare(this.value, anotherInteger.value);
}
```

先用Lambda实现，

```
List<Integer> list = Arrays.asList(1000, 2000, 3000);

// 某个对象的成员方法
Integer num = 2000;
List<Integer> compareList = CollectionUtil.convert(list, i -> num.compareTo(i));
System.out.println(compareList);// [1, 0, -1]
```

与前面类似，这里Lambda的代码块中，依然只有对`num.compareTo(i)`的调用，所以可以简写。但是，需要注意的是，这次方法的调用者不是集合的元素，而是一个外部的局部变量`num`，因此不能使用`Integer::compareTo`，因为这样是无法确定方法的调用者。要指定调用者，需要用`对象::方法名`的方式:

```
// 某个对象的成员方法
Integer num = 2000;
List<Integer> compareList = CollectionUtil.convert(list, num::compareTo);
System.out.println(compareList);// [1, 0, -1]
```

构造函数引用

最后一个场景：把集合中的数字作为毫秒值，构建出`Date`对象并放入集合，这里我们就需要用到Date的构造函数:

```
/**
  * @param   date   the milliseconds since January 1, 1970, 00:00:00 GMT.
  * @see     java.lang.System#currentTimeMillis()
  */
public Date(long date) {
    fastTime = date;
}
```

我们可以接收集合中的每个元素，然后把元素作为`Date`的构造函数参数:

```
// 将数值类型集合，转为Date类型
List<Date> dateList = CollectionUtil.convert(list, i -> new Date(i));
// 这里遍历元素后需要打印，因此直接把println作为方法引用传递了
dateList.forEach(System.out::println);
```

上面的Lambda表达式实现方式，代码体只有`new Date()`一行代码，因此也可以采用方法引用进行简写。但问题是，构造函数没有名称，我们只能用`new`关键字来代替：

```
// 构造方法
List<Date> dateList = CollectionUtil.convert(list, Date::new);
dateList.forEach(System.out::println);
```

注意两点：

- 上面代码中的System.out::println 其实是 指定对象System.out的非静态方法println的引用
- 如果构造函数有多个，可能无法区分导致传递失败

接口的默认方法和静态方法

Java 8使用两个新概念扩展了接口的含义：默认方法和静态方法。

默认方法

默认方法使得开发者可以在 不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不强制那些实现了该接口的类也同时实现这个新加的方法。

默认方法和抽象方法之间的区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方法会被接口的实现类继承或者覆写，例子代码如下：

```
private interface Defaulable {
    // Interfaces now allow default methods, the implementer may or
    // may not implement (override) them.
    default String notRequired() {
        return "Default implementation";
    }
}

private static class DefaultableImpl implements Defaulable {
}

private static class OverridableImpl implements Defaulable {
    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}
```

Defaulable接口使用关键字default定义了一个默认方法notRequired()。DefaultableImpl类实现了这个接口，同时默认继承了这个接口中的默认方法；OverridableImpl类也实现了这个接口，但覆写了该接口的默认方法，并提供了一个不同的实现。

静态方法

Java 8带来的另一个有趣的特性是在接口中可以定义静态方法，我们可以直接用接口调用这些静态方法。例子代码如下：

```
private interface DefaulableFactory {
    // Interfaces now allow static methods
    static Defaulable create( Supplier< Defaulable > supplier ) {
        return supplier.get();
    }
}
```

下面的代码片段整合了默认方法和静态方法的使用场景：

```
public static void main( String[] args ) {
    // 调用接口的静态方法，并且传递DefaultableImpl的构造函数引用来构建对象
    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );
    System.out.println( defaulable.notRequired() );
    // 调用接口的静态方法，并且传递OverridableImpl的构造函数引用来构建对象
    defaulable = DefaulableFactory.create( OverridableImpl::new );
    System.out.println( defaulable.notRequired() );
}
```

这段代码的输出结果如下：

```
Default implementation
Overridden implementation
```

由于JVM上的默认方法的实现在字节码层面提供了支持，因此效率非常高。默认方法允许在不打破现有继承体系的基础上改进接口。该特性在官方库中的应用是：给`java.util.Collection`接口添加新方法，如`stream()`、`parallelStream()`、`forEach()`和`removeIf()`等等。

尽管默认方法有这么多好处，但在实际开发中应该谨慎使用：在复杂的继承体系中，默认方法可能引起歧义和编译错误。如果你想了解更多细节，可以参考官方文档。

Optional

Java应用中最常见的bug就是空值异常。

`Optional`仅仅是一个容器，可以存放T类型的值或者`null`。它提供了一些有用的接口来避免显式的`null`检查，可以参考Java 8官方文档了解更多细节。

接下来看一点使用Optional的例子：可能为空的值或者某个类型的值：

```
Optional< String > fullName = Optional.ofNullable( null );
System.out.println( "Full Name is set? " + fullName.isPresent() );
System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );
System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
```

如果`Optional`实例持有一个非空值，则`isPresent()`方法返回`true`，否则返回`false`；如果`Optional`实例持有`null`，`orElseGet()`方法可以接受一个lambda表达式生成的默认值；`map()`方法可以将现有的`Optional`实例的值转换成新的值；`orElse()`方法与`orElseGet()`方法类似，但是在持有null的时候返回传入的默认值，而不是通过Lambda来生成。

上述代码的输出结果如下：

```
Full Name is set? false
Full Name: [none]
Hey Stranger!
```

再看下另一个简单的例子：

```
Optional< String > firstName = Optional.of( "Tom" );
System.out.println( "First Name is set? " + firstName.isPresent() );
System.out.println( "First Name: " + firstName.orElseGet( () -> "[none]" ) );
System.out.println( firstName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
System.out.println();
```

这个例子的输出是：

```
First Name is set? true
First Name: Tom
Hey Tom!
```

如果想了解更多的细节，请参考官方文档。

Streams

新增的Stream API（java.util.stream）将生成环境的函数式编程引入了Java库中。这是目前为止最大的一次对Java库的完善，以便开发者能够写出更加有效、更加简洁和紧凑的代码。

Steam API极大得简化了集合操作（后面我们会看到不止是集合），首先看下这个叫Task的类：

```
public class Streams  {
    private enum Status {
        OPEN, CLOSED
    };

    private static final class Task {
        private final Status status;
        private final Integer points;

        Task( final Status status, final Integer points ) {
            this.status = status;
            this.points = points;
        }

        public Integer getPoints() {
            return points;
        }

        public Status getStatus() {
            return status;
        }

        @Override
        public String toString() {
```

```
            return String.format( "[%s, %d]", status, points );
        }
    }
}
```

Task类有一个points属性，另外还有两种状态：OPEN或者CLOSED。现在假设有一个task集合：

```
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),
    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
);
```

首先看一个问题：在这个task集合中一共有多少个OPEN状态的？计算出它们的points属性和。在Java 8之前，要解决这个问题，则需要使用foreach循环遍历task集合；但是在Java 8中可以利用steams解决：包括一系列元素的列表，并且支持顺序和并行处理。

```
// Calculate total points of all active tasks using sum()
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter( task -> task.getStatus() == Status.OPEN )
    .mapToInt( Task::getPoints )
    .sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );
```

运行这个方法的控制台输出是：

```
Total points: 18
```

这里有很多知识点值得说。首先，`tasks`集合被转换成`steam`表示；其次，在`steam`上的`filter`操作会过滤掉所有`CLOSED`的`task`；第三，`mapToInt`操作基于`tasks`集合中的每个`task`实例的`Task::getPoints`方法将`task`流转换成`Integer`集合；最后，通过`sum`方法计算总和，得出最后的结果。

在学习下一个例子之前，还需要记住一些steams（点此更多细节）的知识点。Steam之上的操作可分为中间操作和晚期操作。

中间操作会返回一个新的steam——执行一个中间操作（例如filter）并不会执行实际的过滤操作，而是创建一个新的steam，并将原steam中符合条件的元素放入新创建的steam。

晚期操作（例如forEach或者sum），会遍历steam并得出结果或者附带结果；在执行晚期操作之后，steam处理线已经处理完毕，就不能使用了。在几乎所有情况下，晚期操作都是立刻对steam进行遍历。

steam的另一个价值是创造性地支持并行处理（parallel processing）。对于上述的tasks集合，我们可以用下面的代码计算所有task的points之和：

```
// Calculate total points of all tasks
final double totalPoints = tasks
   .stream()
   .parallel()
   .map( task -> task.getPoints() ) // or map( Task::getPoints )
   .reduce( 0, Integer::sum );

System.out.println( "Total points (all tasks): " + totalPoints );
```

这里我们使用parallel方法并行处理所有的task，并使用reduce方法计算最终的结果。控制台输出如下：

```
Total points (all tasks) : 26.0
```

对于一个集合，经常需要根据某些条件对其中的元素分组。利用steam提供的API可以很快完成这类任务，代码如下：

```
// Group tasks by their status
final Map< Status, List< Task > > map = tasks
    .stream()
    .collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );
```

控制台的输出如下：

```
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}
```

最后一个关于tasks集合的例子问题是：如何计算集合中每个任务的点数在集合中所占的比重，具体处理的代码如下：

```
// Calculate the weight of each tasks (as percent of total points)
final Collection< String > result = tasks
    .stream()                                       // Stream< String >
    .mapToInt( Task::getPoints )                    // IntStream
    .asLongStream()                                 // LongStream
    .mapToDouble( points -> points / totalPoints )  // DoubleStream
    .boxed()                                        // Stream< Double >
    .mapToLong( weigth -> ( long )( weigth * 100 ) ) // LongStream
    .mapToObj( percentage -> percentage + "%" )     // Stream< String>
    .collect( Collectors.toList() );                // List< String >

System.out.println( result );
```

控制台输出结果如下：

```
[19%, 50%, 30%]
```

最后，正如之前所说，Steam API不仅可以作用于Java集合，传统的IO操作（从文件或者网络一行一行得读取数据）可以受益于steam处理，这里有一个小例子：

```
final Path path = new File( filename ).toPath();
try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) {
    lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );
}
```

Stream的方法`onClose()`返回一个等价的有额外句柄的Stream，当Stream的`close()`方法被调用的时候这个句柄会被执行。Stream API、Lambda表达式还有接口默认方法和静态方法支持的方法引用，是Java 8对软件开发的现代范式的响应。

并行数组

Java8版本新增了很多新的方法，用于支持并行数组处理。最重要的方法是`parallelSort()`，可以显著加快多核机器上的数组排序。下面的例子论证了parallexXxx系列的方法：

```
package com.javacodegeeks.java8.parallel.arrays;

import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class ParallelArrays {
    public static void main( String[] args ) {
        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,
            index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();

        Arrays.parallelSort( arrayOfLong );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();
    }
}
```

上述这些代码使用parallelSetAll()方法生成20000个随机数，然后使用parallelSort()方法进行排序。这个程序会输出乱序数组和排序数组的前10个元素。上述例子的代码输出的结果是：

```
Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378
Sorted: 39 220 263 268 325 607 655 678 723 793
```