**1.3.1 lambda与函数式**

在响应式编程中，lambda与函数式的出镜率相当高，以至于网上经常有朋友直接用“函数响应式编程”用在“响应式编程”的介绍中。这两个词的异同一直存在争议，其区别虽然不像“JavaScript与Java”、“雷锋塔与雷峰”那么大，但随便混用还是会显得非常不专业：

* 函数响应式编程的重点在于“函数式”的语言特性，这个概念在二十年前就[盖棺定论](http://conal.net/papers/icfp97/" \t "_blank)了。
* 响应式编程的重点在于“基于事件流”的异步编程范式，由不断产生的数据/时间来推动逻辑的执行。

本系列文章讨论的都是“响应式编程”，关于“函数响应式编程”，你就当没听过，并谨慎地使用它就好了。

**1.3.1.1 lambda表达式**

书回正传，为什么响应式编程中会经常用到lambda与函数式呢？不知你对1.1.3节的一段伪代码是否还有印象：

cartEventStream

// 分别计算商品金额

.map(cartEvent -> cartEvent.getProduct().getPrice() \* cartEvent.getQuantity())

...

cartEventStream是一个数据流，其中的元素就是一个一个的cartEvent，map方法能够对cartEvent进行“转换/映射”，这里我们将其转换为double类型的金额。

除了转换/映射（map）外，还有过滤（filter）、提供（supply）、消费（consume）等等针对流中元素的操作**逻辑/策略**，而逻辑/策略通常用方法来定义。

在Java 8之前，这就有些麻烦了。我们知道，Java是面向对象的编程语言，除了少数的原生类型外，一切都是对象。用来定义逻辑/策略的方法不能独立存在，必须被包装在一个对象中。比如我们比较熟悉的Comparator，其唯一的方法compare表示一种比较策略，在使用的时候，需要包装在一个对象中传递给使用该策略的方法。举例说明（[源码](https://github.com/get-set/get-reactive/blob/master/snacks/src/test/java/com/getset/Test_1_3.java#L21)）：

@Test

public void StudentCompareTest() {

@Data @AllArgsConstructor class Student { // 1

private int id;

private String name;

private double height;

private double score;

}

List<Student> students = new ArrayList<>();

students.add(new Student(10001, "张三", 1.73, 88));

students.add(new Student(10002, "李四", 1.71, 96));

students.add(new Student(10003, "王五", 1.85, 88));

class StudentIdComparator<S extends Student> implements Comparator<S> { // 2

@Override

public int compare(S s1, S s2) {

return Integer.compare(s1.getId(), s2.getId());

}

}

students.sort(new StudentIdComparator<>());

System.out.println(students);

}

1. @Data和@AllArgsConstructor是lombok提供的注解，能够在编译的字节码中生成构造方法、getter/setter、toString等方法。依赖如下：
2. <dependency>
3. <groupId>org.projectlombok</groupId>
4. <artifactId>lombok</artifactId>
5. <version>1.16.20</version>
6. </dependency>

注：本节及以后，关于maven的依赖，可自行至[maven搜索库](http://search.maven.org" \t "_blank)中查询新的合适版本。

1. StudentIdComparator中固化了一种针对Student.id的比较策略，当对students进行排序的时候，将StudentIdComparator的对象传给sort方法。输出顺序如下：

[Student(id=10001, name=张三, height=1.73, score=88.0), Student(id=10002, name=李四, height=1.71, score=96.0), Student(id=10003, name=王五, height=1.85, score=88.0)]

假设这时候我们需要对学生的身高或分数进行排序，再定义Comparator的实现类有些麻烦了，而且没必要，“传统”的简化方式是直接传入匿名内部类：

students.sort(new Comparator<Student>() {

@Override

public int compare(Student s1, Student s2) {

return Double.compare(s1.getHeight(), s2.getHeight());

}

});

但其实，我们会发现，无论哪种比较策略，只有compare方法内的代码发生变化，也就是说sort方法关心的只是传入的两个参数Student s1, Student s2以及返回的结论return Double.compare(s1.getHeight(), s2.getHeight())这一句比较策略，何不只保留它们呢？

students.sort((Student s1, Student s2) -> {return Double.compare(s1.getHeight(), s2.getHeight());});

这样看起来代码就少多了。其中(Student s1, Student s2) -> {return Double.compare(s1.getHeight(), s2.getHeight())}就是lambda表达式，lambda表达式的语法如下：

(type1 arg1, type2 arg2...) -> { body }

* 1->前后分别表示参数和方法体。从代码编写方式上来说，这就可以算作是“函数式”编程范式了，因为我们传给sort的是一个lambda表达式的形式定义的“函数”，这个“函数”有输入和输出，在开发者看起来是赤裸裸的，没有使用对象封装起来的。

“函数式”编程范式的核心特点之一：函数是”一等公民”。   
所谓”一等公民”（first class），指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

但也仅仅是“看起来”是“函数式”的了，Java终究是面向对象的语言，List.sort的方法定义仍然是接受一个Comparator对象作为参数的。但是一定要纠结Java是不是纯正的函数式语言吗？没这个必要，实用至上嘛。

既然如此，问题来了，sort是如何将这个lambda“看做”一个Comparator对象的呢？

不难发现，Comparator接口仅有一个抽象方法，因此sort也就不难“推断”lambda所定义的输入参数和方法体表示的正是这个唯一的抽象方法compare。

**1.3.1.2 函数式接口**

像Comparator这样的只有一个抽象方法的接口，叫做函数式接口（Functional Interface）。与Comparator类似，其他函数式接口的唯一的抽象方法也可以用lambda来表示。

我们看一下Comparator的源码，发现其多了一个@FunctionalInterface的注解，用来表明它是一个函数式接口。标记了该注解的接口有且仅有一个抽象方法，否则会报编译错误。

再看一下其他的仅有一个抽象方法的接口，比如Runnable和Callable，发现也都在Java 8之后加了@FunctionalInterface注解。对于Runnable来说，接口定义如下：

@FunctionalInterface

public interface Runnable {

public abstract void run();

}

不难推测，其lambda的写法应该是 () -> { body }，它不接收任何参数，方法体中也无return返回值，用起来像这样：

new Thread(() -> {doSomething();});

* 1此外，随lambda一同增加的还有一个[java.util.function包](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-frame.html" \t "_blank)，其中定义了一些常见的函数式接口的。比如：
* Function，接受一个输入参数，返回一个结果。参数与返回值的类型可以不同，我们之前的map方法内的lambda就是表示这个函数式接口的；
* Consumer，接受一个输入参数并且无返回的操作。比如我们针对数据流的每一个元素进行打印，就可以用基于Consumer的lambda；
* Supplier，无需输入参数，只返回结果。看接口名就知道是发挥了对象工厂的作用；
* Predicate，接受一个输入参数，返回一个布尔值结果。比如我们在对数据流中的元素进行筛选的时候，就可以用基于Predicate的lambda；
* …

**1.3.1.3 简化的lambda**

以lambda作为参数的方法能够推断出来lambda所表示的是哪个函数式接口的那个抽象方法。类似地，编译期还可以做更多的推断。我们再回到最初的Comparator的例子并继续简化如下lambda表达式：

students.sort((Student s1, Student s2) -> {return Double.compare(s1.getHeight(), s2.getHeight());});

1）首先，传入的参数类型是可以推断出来的。因为students是以Student为元素的数组List<Student>，其sort方法自然接收Comparator<? super Student>的对象作为参数，这一切都可以通过泛型约束。

students.sort((s1, s2) -> {return Double.compare(s1.getHeight(), s2.getHeight());});

2）如果只有一个return语句的话，return和方法体的大括号都可以省略(compare方法的返回值就是lambda返回值）：

students.sort((s1, s2) -> Double.compare(s1.getHeight(), s2.getHeight()));

3）注意到，Comparator接口还提供了丰富的静态方法，比如：

public static<T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor) {

Objects.requireNonNull(keyExtractor);

return (Comparator<T> & Serializable)

(c1, c2) -> Double.compare(keyExtractor.applyAsDouble(c1), keyExtractor.applyAsDouble(c2));

}

这个方法为我们包装好了Double.compare。它接收一个返回类型为Double的函数式接口ToDoubleFunction<? super T>，可以看做是Function<? super T, Double>，用lambda表示的话就是student -> student.getHeight()。

因此，我们的sort方法又可以写作：

students.sort(Comparator.comparingDouble((student) -> student.getHeight()));

其一，对于只有一个参数的lambda来说，参数外边的小括号可以省略：

students.sort(Comparator.comparingDouble(student -> student.getHeight()));

其二，对于仅有一个方法调用的lambda方法体来说，通常又可以用类::方法进一步简化，以上代码又可以进一步简化为：

students.sort(Comparator.comparingDouble(Student::getScore));

这里是调用参数所代表对象的某个方法，与之类似的还有比如：

* string -> System.out.println(string)，可以简化为System.out::println，这里是将参数作为System.out::println的参数了；
* () -> new HashMap<>()，可以简化为HashMap::new，这里没有参数，也可以进行简化。

使用类::方法这种写法是不是更加有函数式的感觉了呢，似乎真是把函数作为参数传递给某个方法了呢~