

java8是java语言历史上变化最大的一个版本，java由纯粹的面向对象语言开始转向函数式编程风格。以前java很大的特点就是传递并操作数据，无论是简单类型还是引用类型都是数据，而函数式编程传递的是行为，把行为（函数）作为参数，在不同的地方传递。

一.lambda表达式

1.初始

```
1 Thread myThread = new Thread(new Runnable() {
2     @Override
3     public void run() {
4         System.out.println("hello,sean");
5     }
6 });
7 myThread.start();
```

这是一个非常典型的匿名内部类，在idea中已经给出提示，这种写法可以用lambda表达式所替代，替代后的结果如下：

```
1 Thread myThread = new Thread(() -> System.out.println("hello,sean"));
```

现在可以先对lambda表达式有个认识，lambda表达式分为三个部分，左边是参数，中间是箭头符号，右边是执行体，在angular里写ts代码中已经使用过很多次了。

```
1 (param1,param2,param3) -> {
2
3 }
```

2.@FunctionalInterface注解（根源）

java8新增，这是一个提醒式注解，本身并没有任何意义，但是这个注解与lambda表达式密切相关，在查看源码会发现有这样一段注释：

```
1 A functional interface has exactly one abstract method
2 Note that instances of functional interfaces can be created with
3 lambda expressions, method references, or constructor references.
```

首先，被@FunctionalInterface标注的接口，有且只有一个抽象方法，换言之如果要实例化就只需要重写一个抽象方法。非常熟悉的线程Runnable接口，和java8新增的Consumer接口都是函数式接口。

其次，函数式接口的实例化可以通过lambda表达式，方法引用，构造引用，现在重点关注lambda表达式创建，以后如果需要一个函数接口作为参数，就可以用lambda表达式来描述这样一个参数：

```
1 list.forEach((Integer i)-> System.out.println(i));
2 //foreach需要的参数是Consumer接口的实例，而Consumer是函数式接口，因此可以
3 //使用lambda表达式来实例化。
```

最后，如果一个接口符合函数式接口的特征（是接口，并且只有一个抽象方法），那么这个函数就是函数式接口，需要标注@FunctionalInterface注解，即使不标注，编译器也会把它当作是函数式接口，但是最好标注一下。

关于这个函数式注解有另外一个值得注意的点，注解有抽象方法，但是这个抽象方法是toString()或者其他覆盖了Object类的方法，那这个不算入抽象方法的个数，原因很简单，接口的实现类必然继承Object类，自然就实现了toString()方法，真正要实现的抽象方法是那种接口定义的自定义方法，这种实现才需要借助lambda表达式。

讲到这里，其实对于lambda表达式会有个认识，之所以对函数式注解有有限制，就是为了当接口有实现类时，只需要重写一个方法，在这种特定情况下，可以使用lambda表达式简单写法，因为lambda表达式生成实例时只需要实现一个抽象方法就可以了。

```
1 Consumer<Integer> consumer = (Integer i)->{
2     System.out.println(i);
3 };
```

这段代码说明了lambda表达式其实就是实现了一个接口，并重写了里面唯一一个抽象方法，就是匿名内部类的简单写法。

做个完整的说明，foreach方法需要一个Consumer类型的参数，里面对集合的每一个参数调用一次accept方法，而我们使用lambda表达式就是实现了Consumer接口并重写了accept方法，把集合里的每一个元素都输出了一次。

Lambda表达式是一个对象，和其他语言不同，其他语言（例如Js）来说函数，而在java中是对象，它依赖于一个特别的对象类型—FunctionInterface！

3.Consumer接口

这是一个函数式接口，这个接口代表了一种操作，这种操作接受单个输入参数，并且不返回结果：

```
1 void accept(T t);
```

这就是所谓的消费，只操作不返回值。

4.方法引用

函数式接口的实例不仅可以用lambda表达式创建，还可以用方法引用来创建：

```
1 list.forEach(System.out::println);
2 //对list里的每一个元素进行一次输出
```

方法引用后面会讲，这里先说一下自己的探索，方法引用必须是这个操作只调用一次方法，没有其他任何操作。否则的话不能使用方法引用的写法。

5.Lambda表达式的类型

既然lambda表达式是对象，那么这个对象到底是什么类型的？

对于一个普通的lambda表达式：

```
1 () -> {};
```

我们无法单独判断它是什么类型，必须通过上下文得知，例如：

```
1 Runnable myRunnable = ()->{};
```

这样就知道lambda表达式是Runnable类型的。其实就是匿名内部类的另一种写法，同样要给出方法实现。

6.初识流

java8在Collection接口中新添加了stream相关的default方法（指接口中的已经实现好了的方法），可以返回一个流，这个流的源就是调用这个方法的集合。

```
1 list.stream().map((item)->item.toUpperCase()).forEach((item)->System.out.println(item));
2 list.stream().map(String::toUpperCase).forEach(System.out::println);
```

幸好在javascript里修炼过，对这段代码有很强的抵抗力了，基本上看过就懂，把list里面的元素全部转换成大写，并逐项输出。

关于方法引用，现在还是不太清楚，毕竟map和forEach不一样，toUpperCase和println也不一样，为什么写法差不多，还得听后面讲才行。

7.Function接口

之前学了Consumer接口，用在forEach里面，而上面的stream.map方法需要的是Function接口，与Consumer接口有一定的差别：

```
1 public interface Function<T, R> {
2     /**
3      * Applies this function to the given argument.
4      *
5      * @param t the function argument
6      * @return the function result
7      */
8     R apply(T t);
9 }
```

作为一个函数式接口，Function接口里唯一的抽象方法是：获得一个参数，并返回一个结果。

```
1 Function<String,String> function = (item)->item.toUpperCase();
2 Function<String,String> function = String::toUpperCase;
3 Function<String,Boolean> function = String::isEmpty;
```

来看一下这种方法引用创建函数式接口实例的代码，toUpperCase并不是一个静态方法，是String类的方法，需要一个字符串对象去调用，又必定返回一个字符串，调用该方法的字符串必定是lambda表达式的第一个参数item，只要告诉java这个方法是哪个类的，java就能推断出参数和返回值是什么类型，就OK了。

Function接口的使用和Consumer差不多，代码如下：

```
1 Function<Integer,Integer> function = (value->value*2);
```

这种传递行为的代码风格和以前预定义行为，调用方法的风格是完全不同的。

8.Lambda基本语法

java8的lambda表达式其实就是一个匿名函数，通过重写接口的抽象方法来达到实例化接口的目的。其作用是传递行为，而不仅仅是传递值。

```
1 (argument)->{body};
2 (arg1,arg2)->{body};
```

```
3 {type1 arg1,type2 arg2}->{body};//对于我个人而言更喜欢把类型写全
```

1.箭头符号把左右两边区分开，左边是参数，右边是方法体，左边的参数类型通常可以省略，编译器会进行类型推断，但是如果希望代码可读性更好，也可以写上；

2.所有参数放在圆括号里，参数之间用，隔开，空的圆括号表示参数为空；

3.当只有一个参数并且其类型可推导，那么（）可以省略；

4.如果函数体body只有一句表达式(expression)，那么花括号{}也可以省略，效果如下：

```
1 list.forEach(item->System.out.println(item));
```

5.匿名函数返回类型与代码块返回类型一致，如果没有则为void：

```
1 Comparator comparator = (String o1,String o2)->{return  
o1.compareTo(o2)};;  
2 Comparator comparator = (String o1,String o2)->o1.compareTo(o2);
```

在这里o1.compareTo(o2)就是方法体的返回类型，不需要自己去写return。

9.Function接口详解

1.compose默认方法

```
1 default <V> Function<V, R> compose(Function<? super V, ? extends T> before  
e) {  
2 Objects.requireNonNull(before);  
3 return (V v) -> apply(before.apply(v));  
4 }
```

它实现的作用是把before这个function实例和this做一个合并，组合成一个新的function函数，before先调用，this后调用，返回的就是这个复合函数。

2.andThen默认方法

```
1 default <V> Function<T, V> andThen(Function<? super R, ? extends V>  
after) {  
2 Objects.requireNonNull(after);  
3 return (T t) -> after.apply(apply(t));  
4 }
```

和上面的compose类似，只是先后顺序有所不同，先调用this函数，再调用after函数，返回一个符合函数。

上面两个default方法实际使用下来还是比较清楚的，就是把两个Function实例进行组合形成复合函数，并且这两个Function实例有先后使用顺序，泛型也必须衔接上，第一个函数的返回值类型必须是第二个函数的参数类型。测试可以直接使用apply，或者使用stream.map测试。

10.BiFunction接口详解

Function接口只能接受一个参数返回一个结果，而BiFunction可以接受两个参数并返回一个结果，是Function接口的特化形式：

```
1 @FunctionalInterface
2 public interface BiFunction<T, U, R> {
3
4     /**
5      * Applies this function to the given arguments.
6      *
7      * @param t the first function argument
8      * @param u the second function argument
9      * @return the function result
10    */
11    R apply(T t, U u);
```

11.Predicate接口详解

```
1 result = persons.stream().filter(person -> person.getAge() >
17).collect(Collectors.toList());
```

这个和javascript里的filter非常相似，把集合中符合某个条件的元素组合成一个新的集合，这里需要强调一点自己现在还不适应的地方，就是我们重写的方法是需要有返回值的，但是在lambda表达式中并不一定需要写return，因为如果lambda表达式的方法体中，只有一个表达式，那么就会直接把这个表达式的结果返回，不需要自己写return了。

在idea中会提示把statement lambda（语句lambda）替换成expression lambda（表达式lambda）。

```
1 @FunctionalInterface
2 public interface Predicate<T> {
3
4     /**
```

```

5  * Evaluates this predicate on the given argument.
6  *
7  * @param t the input argument
8  * @return {@code true} if the input argument matches the predicate,
9  * otherwise {@code false}
10 */
11 boolean test(T t);

```

接口和抽象方法如下，我们就是在重写里面的test方法。Predicate接口通常被使用于对集合的filter过滤操作，和javascript一样。

函数式编程给我们带来了什么？

比如对一个list进行不同条件的筛选，奇数，偶数，大于5等等，在传统的面向对象编程中，我们必须定义不同的方法去实现这些功能，但是在函数式编程中，我们可以只定义一个概括的filter方法，把filter的具体逻辑交给方法的调用者去提供。

1.and默认方法

```

1 default Predicate<T> and(Predicate<? super T> other) {
2     Objects.requireNonNull(other);
3     return (t) -> test(t) && other.test(t);
4 }

```

和Function接口里的andThen，after有点类似，就是把两个Predicate进行and组合，生成一个新的Predicate对象，必须是两个条件同时满足才返回true。

2.or默认方法

```

1 default Predicate<T> or(Predicate<? super T> other) {
2     Objects.requireNonNull(other);
3     return (t) -> test(t) || other.test(t);
4 }

```

把两个Predicate进行or组合。值得一提的是以上的or和and都是短路操作。

3.isEqual静态方法

```

1 static <T> Predicate<T> isEqual(Object targetRef) {

```



```

2  return (null == targetRef)
3  ? Objects::isNull
4  : object -> targetRef.equals(object);
5  }

```

传一个对象，返回一个Predicate，这个Predicate用于得到一个equal方法。用起来有点怪，不会经常用。

12.Supplier接口详解

```

1  @FunctionalInterface
2  public interface Supplier<T> {
3
4      /**
5       * Gets a result.
6       *
7       * @return a result
8       */
9      T get();
10 }

```

不接受任何参数，同时返回一个结果。

到此为止函数式接口的介绍告一段落，在后面讲解stream流的过程中会得到非常广泛的应用。

13.Optional

Optional主要用于规避空指针异常，提高代码的健壮性。

1.创建Optional对象

有三种创建Optional对象的方式：

```

1  public static<T> Optional<T> empty() {
2      Optional<T> t = (Optional<T>) EMPTY;
3      return t;
4  }
5  public static <T> Optional<T> of(T value) {
6      return new Optional<>(value);
7  }
8  public static <T> Optional<T> ofNullable(T value) {
9      return value == null ? empty() : of(value);

```



```
10 }
```

第一种方式empty()会创建一个空的容器对象，value为null；

第二种方式of()会接受一个必须不为null的对象，生成一个容器对象；

第三种方式ofNullable()接受一个可能为null也可能不为null的对象，生成一个容器对象，我现在比较习惯用第三种。

2.如何使用Optional对象

```
1 if(optional.isPresent()){
2     optional.get();
3 }
```

这种对Optional的使用方法的思路依然和以前的非空校验一样，并不推荐，推荐使用函数式编程风格：

```
1 optional.ifPresent(str -> System.out.println(str));
2 //如果容器内的值不为null，进行操作；
3
4 optional.orElse("wrong");
5 //如果容器内的值不为null，返回值，否则返回"wrong"
```

```
1 System.out.println(optional.orElseGet(()->"world"));
2 optional.map(teacher-
>teacher.getStudents()).orElse(Collections.emptyList());
```

最后一句代码的意思是根据teacher对象获取students集合，如果teacher里的students为null，就返回空集合。

接下来是一段我自己研究的代码：

```
1 public boolean login(User user) {
2     return Optional.ofNullable(user).filter(userFilter -> userFilter.getUserName() == userFilter.getPassword()).isPresent();
3 }
```

这段代码通过一行完成了非空校验加登录判断，使用了option的filter方法来进行判断，对option有三种操作，Consumer，Function，Predicate，对应的方法名为ifPresent()，map()，filter()，看情况使用。

对于Optional对象要有一个直观的印象，它是一个容器对象，里面有一个value属性，它有可能为空，也可能不为空，围绕着这一点进行操作。

14.方法引用

方法引用是lambda表达式的一个语法糖，在有些时候，lambda表达式恰好有一个方法可以做到，这时就可以用方法引用的写法来代替。

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

我们可以将方法引用看作是一个函数指针，指向了一个函数（方法）。对list里的每一个元素调用System.out.println()这个方法。

方法引用共分为4类：

1.类名::静态方法名

```
1 students.sort((Student s1, Student s2) -> s1.getAge() - s2.getAge());
2 students.sort((Student s1, Student s2) -> Student.compareByAge(s1,s2));
3 students.sort(Student::compareByAge);
```

sort方法需要接受一个Comparator实例，抽象方法为接受两个同类型参数，返回int。而当我们定义的静态方法恰好满足这一入参和出参的条件，就可以使用方法来代替，简化代码书写。

2.对象名::实例方法名

```
1 StudentCompare studentCompare = new StudentCompare();
2 students.sort(studentCompare::compareStudent);
```

与上面的静态方法调用非常类似的，在StudentCompare类中定义了一个方法，接受两个student对象，返回他们的年龄差。其入参和出参和Comparator抽象方法一样，所以也可以用方法来代替。

上面这两种方法引用的形式，有一个共同特点，方法的入参出参和抽象方法的入参出参是一样的，这时本来用lambda表达式去做的事情，可以简单地用方法引用的写法代替。

3.类名::实例方法名（比较难以理解）

上面两种方法引用的方式都要求方法的入参和出参与抽象方法完全一样，但是这种方法就不一样了。

```
1 public int compareByAge2(Student s1) {
2     return this.getAge() - s1.getAge();
3 }
4 students.sort(Student::compareByAge2);
```

sort方法需要接受Comparator类型实例，Comparator抽象方法需要接受两个对象，返回int，而我们定义的compareByAge2只接受一个对象。这里运行的机制是：lambda表达式的第一个参数负责调用方法，后面所有参数作为方法的入参。

Comparator实例写成lambda表达式就是(s1,s2)->(s1.getAge()-s2.getAge())，在把这个lambda表达式替换成方法引用的过程中，会把s1作为方法的调用者，s2作为方法的参数，从而达到方法调用的结果。

4.类名::new

实际上就是调用这个类的构造方法创建对象。

```
1 public Student(String name) {  
2     this.name = name;  
3 }  
4 List<Student> students = list.stream().map(Student::new).collect(Collectors.toList());
```

这里把list集合里每一个string作为构造方法的参数传递，返回students集合。

关于方法引用，并没有必要研究的太深，归根到底只是lambda表达式的特殊形式，掌握了lambda表达式就足够了。学习方法引用主要还是为了看懂别人的代码。

二.Stream流

在java8种，Stream流是一个与Lambda表达式相伴相生的极其重要的概念。借助Stream流，我们可以对一个集合进行一连串的串行或并行操作，如果用传统的集合操作方法会写出非常繁琐的代码，而如果用Stream+Lambda的方式，代码会非常简洁。

对流的操作并不会改变数据源，而是生成一个新的结果。很多流操作都是延迟操作。

1.初始流

1.流的组成

流由以下三部分组成：

- 1.源，指流的数据源，比如集合，数组；
- 2.零个或多个中间操作，会把一个流转换成另一个流；
- 3.终止操作。

2.流操作的分类

- 1.惰性求值，流可以用链式调用函数的方式去操作，例如：

```
1 stream.xxx().yyy().count();
```

其中，xxx()和yyy()只会在调用count()（终止操作）才会执行，如果没有count()这个终止操作，那么xxx()和yyy()不会执行，这就有点像angular那里，发请求必须subscribe才能生效。惰性求值其实就是中间操作。

- 2.及早求值，其实就是终止操作。

2.创建流对象

1.调用Stream的静态方法

```
1 Stream stream = Stream.of(1, 2, 3, 4, 5);
```

2.用数组创建Stream对象

```
1 String[] arr = {"1", "2", "3", "4", "5"};
2 Stream stream2 = Stream.of(arr);
3 Stream stream3 = Arrays.stream(arr);
```

3.用集合创建Stream对象（最常用）

```
1 // 用集合创建Stream对象
2 List<String> list = Arrays.asList("1", "2", "3", "4", "5");
3 Stream stream4 = list.stream();
```

3.操作流对象

```
1 List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
2 int sum = list.stream().map(item -> item *
2).mapToInt(Integer::intValue).sum();
3 System.out.println(sum);
```

把数字集合的每一项*2，并求和。中间经历了把普通Stream转换成IntStream的过程。

终止操作里有一个重要方法，把stream转换成list集合，代码如下：

```
1 List<String> streamList = stream2.collect(Collectors.toList());
2 streamList = stream2.collect(() -> new ArrayList<>(), (list, item) -> list.add(item), (list1, list2) -> list1.addAll(list2));
3 List<String> streamList = stream2.collect(Collectors.toCollection(LinkedList::new));
```

第一行是简便写法（得到ArrayList集合），第二行是原理级别的写法，collect方法接受三个参数，第一个参数定义返回的是什么结果（这里定义了一个ArrayList集合作为结果容器），第二个参数定义的是对stream里的每一个元素做什么操作（这里让stream的每一个元素都放进结果容器中），第三个参数是考虑到如果在并行流中可能会产生多个结果容器，对这些结果容器进行合并（这里让一个list去添加另一个list）。

第三行是一种可以生成我们想要的集合类型的转换方法，不仅List，还可以形成Set集合。