Java8 之 lambda 表达式

一、 简介

自 2014 年 sun 公司发布 java8 以来，已经经历了 6 个年头，其新特性经历了充分的考验，得到了广大社区的狂热追捧。Java8 作为 java 迭代版本史上一次重大的更新，堪称里程碑。其彻底改变了程序员的思维，由以前的命令式编程转变为功能式编程，极大的加快了开发效率。

优点：
【1】 命令式编程转变为功能式编程，加快开发效率
【2】 代码简介，通俗易懂，易于调试
【3】 丰富的集合操作符，集合的操作变得不再困难和冗余
【4】 优秀的异常处理机制，处理异常变得不再困难
【5】 对集合的并发操作提供支持，降低并发问题
【6】 符合目前主流响应式编程方式和思维，社区量庞大

任何一门开发语言或者思维，有其独特的优点，当然无可避免的也有一些瑕疵，但是仍然可以从一定方向进行避免。
缺点：
【1】 知识系统庞大，入门门槛高，初学者往往找不到使用场景
【2】 对并发操作支持不是很灵活，不能在集合处理过程中更换线程
【3】 Java8 中的 lambda 本质上一种语法糖，针对于集合的语法糖

二、 前言

学习 lambda 表达式你需要了解：
1、java 泛型，集合，接口等
2、reactor 模型。

除此之外，如能了解以下框架，则能更近一步加深对 lambda 表达式的理解：
1、webflux
2、rxjava

三、 概念

概念往往是枯燥且乏味的，请结合后面的实际例子，来回查看概念，加深对概念的理解。
1、Lambda 表达式中箭头前面的是参数，箭头后面的是返回值或者是操作语句。多个参数用括号包起来，并用逗号分割。多条操作语句时加花括号，每条语句用分号分割。如：

```
(a,b,c)->{
    System.out.println(a);
    System.out.println(b);
```

```
    System.out.println(c);
    return a+b+c; //返回值可以省略，根据函数定义的功能来决定
};
```

表达的内容是：一个有三个参数 a,b,c 的匿名函数，函数的内容是依次输出 a,b,c 然后返回三个数之和。

2、lambda 表达式的出现是为了解决事件中进行回调问题，许多回调接口中有且仅有一个回调函数，过多的回调接口导致类文件过多，而这些类本身并没有多大意义。

比如线程接口 Runnale，用 lambda 表达式我们可以这样新建线程：

```
//新开线程
public void newRunnableLambda() {
    new Thread(() -> {
        System.out.println("用 lambda 表达式代替匿名内部类");
    }).start();
}
```

当处理操作仅仅只有一条语句时，可以去掉花括号和分号，此时结果变为：

```
//新开线程
public void newRunnableLambda() {
    new Thread(() ->System.out.println("用 lambda 表达式代替匿名内部类")).start();
}
```

3、lambda 表达式只能同一种方向推断出同一种类型的数据，具体可参见 Stream(概念第五章，用法第七章)，且推断的结果可以用任何字符表示。你可以用 a，b，c 这种单个无意义的字符，也可以用有意义的单词。不过目前社区比较主流的写法都是用一个字符表示，也推荐使用。

四、 lambda 表达式原型

像学习英文一样，英文单词分原型，过去型，三单型等等，java 中的 lambda 表达式也有原型，许多二元，三元的 lambda 都是由一元的转换而来。掌握 lambda 表达式原型是学习 lambda 的必经途径。函数式接口可以用 lambda 来表示，可以简单的理解为函数式接口就是 lambda 表达式，如后文中有提到，希望读者明白。

1、自定义函数式接口原型

Java 中的 lambda 表达式比较鸡肋，本质上是一种语法糖，需要先定义，然后才能使用。任何接口加上@FunctionalInterface 都可以使之成为一个函数式接口，可以用 lambda 表达式表示。后来经过改版，即使不加该注解，仍然可以推断出。但值得注意的是不管加不加注解，函数式接口中都必须只能有一个抽象方法（默认方法和静态方法可以有多个）。

举几个例子：Runnanle，Comparable，ActionListe 等等这些接口中只有一个抽

象方法，那么以后我们的写法就可以改为：

```
private void functionLambda() {
    //创建一个线程接口
    Runnable runnable = () -> System.out.println("this is async logic");
    //点击事件回调函数
    ActionListener actionListener = e -> System.out.println(e);
    //创建一个恒等比较器
    Comparable<Object> comparable = o -> 0;
}
```

2、系统自带函数式接口

四大内置核心函数式接口（原型）：

函数式接口	参数类型	返回类型	用途
Consumer<T> 消费型接口	T	void	对类型为T的对象应用操作，包含方法： void accept(T t)
Supplier<T> 供给型接口	无	T	返回类型为T的对象，包含方法：T get();
Function<T, R> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法：R apply(T t);
Predicate<T> 断定型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回 boolean 值。包含方法 boolean test(T t);

其他接口（补充型）：

www.atguigu.com

其他接口

函数式接口	参数类型	返回类型	用途
BiFunction<T, U, R>	T, U	R	对类型为 T, U 参数应用操作，返回 R 类型的结果。包含方法为 R apply(T t, U u);
UnaryOperator<T> (Function子接口)	T	T	对类型为T的对象进行一元运算，并返回T类型的结果。包含方法为 T apply(T t);
BinaryOperator<T> (BiFunction 子接口)	T, T	T	对类型为T的对象进行二元运算，并返回T类型的结果。包含方法为 T apply(T t1, T t2);
BiConsumer<T, U>	T, U	void	对类型为T, U 参数应用操作。包含方法为 void accept(T t, U u)
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T	int long double	分别计算int、long、double、值的函数
IntFunction<R> LongFunction<R> DoubleFunction<R>	int long double	R	参数分别为int、long、double 类型的函数

说明：

1、基本的原型一共 4 个，分为消费型接口，表现为：

```
t->{   //或者(t)
    System.out.println(t);
}
```

表示这是一个消费函数，不返回任何数据，lambda 表达式到此处时表示推断结束。

供给型接口，表现为：

```
() ->new Random().nextInt();
```

表示这是一个提供函数，生成一个指定数据，一般此函数为 lambda 表达式的起点。

函数型接口，表现为：

```
(a)->{
    //对 a 进行一系列函数操作，然后返回一个结果
    int i = Integer.parseInt(a);
    i++;
    return i;
}
```

表示这是一个转换函数，需要对数据进行转换，一般此函数为 lambda 表达式的中间状态。

断定型接口，表现为：

```
(a)->{
    return a>0;
}
```

表示这是一个判断函数，可以处在 lambda 表达式中的任何地方，返回值为布尔类型，会过滤掉为 false 的数据。如上面的案列，将会过滤掉 a<0 的数据，仅留下 a>0 的数据进行后续的操作。

2、补充型接口为多元 lambda 接口，原理和单元接口一样，只是参数个数有所变化。

3、针对原型或者多元函数式接口中的泛型参数，没有必要专门去记，idea 一般会有提示的，但是为了能帮助大家更好的掌握 lambda 表达式，我自己总结了一些：

【1】 T = type = 类型，参数一

【2】 U = T = type = 类型，参数二

【3】 R = result = 结果，返回值

【4】 Bi = binary = 二元，表示有两个不同类型参数

【5】 Un = unary = 一元 表示有一个参数

【6】 操作者 Operator 是对参数本身进行转换，然后返回转换结果。其本身也是二元函数。但是它是特殊的二元函数，参数类型和返回值类型一样。

【7】 To = 到，表示返回值结果。同理可以理解 IntFunction，表示参数为 int

五、 基本 API

1、Collection

集合，其实现类包含常用的 list，set，map，queue，stack 等等

2、Collections

集合工具类，包含采用装饰器模式处理的同步集合，生成集合的工厂方法，集合的二分查找法等等

3、Collector

收集器，java8 新 api，自定义收集器时使用，将流按照自定义方式收集起来。一般此类用的不多，多用收集器工具类 Collectors(具体用法见第七章)。收集器包含提供者，收集者，合并者，转换者。其中合并者专用于并发操作时的处理方案，其余三个见 lambda 原型词条。

4、Collectors（具体用法见第七章）

收集器工具类，java8 新 api，包含了众多收集器，比如 list 收集器，set 收集器，map 收集器，分组收集器，分区收集器，数据统计收集器等等，该类为相当重要的类，灵活使用该类加极大加快开发速度。

5、collect

动词，收集。Java8 新 api，多用于在流操作结束之前，对流进行收集的操作，是一个函数。

6、Stream

流，java8 新 api，任何对象都可以拥有自己的流。请区别 IO 中的流和此处的流。此处的流是将数据按照流式的方向进行操作，只能从源点到终点，就像流水一样，不能倒流，也不能重复流（如果流已经被收集了，不能再对该流进行操作，必须重新创建流）。而 IO 中的流是对具体的文件或者网络进行数据读取，逻辑上称读取的数据为流数据，其方向可以从源点到终点，也可从终点到源点。Java8 采用反应式编程，是 reactor 模型的一种应用。

(1)、流的获取

流的获取方式有三种：

A、 任何对象都可以使用 Stream.of(T...t)或者 Stream.ofNullable(T t)获取它对应的流，其中第二个在获取到的流为空时不会抛异常。注意采用这种方式获取到的流是它本身的流。如果是集合的话，获取的是这个集合的流，并不是集合里面数据的流。如果要获取集合里面数据的流，请用第二种方式。见下面案列：

```
private void streamTest(){
    //获取字符串的流
    Stream<String> stringStream = Stream.of("111", "222");
    //获取整形数据的流
    Stream<Integer> integerStream = Stream.of(222);
    //获取链表的流
    Stream<List<String>> listStream = Stream.of(Arrays.asList("123", "235"));
}
```

B、 获取集合中数据的流，可以使用 Collection.stream()来获取集合里面内容的流。

见下面案列：

```
private void streamContentTest(){
    //获取到的是集合里面数据的流，比如此处集合中的数据是字符串，此处获取到的就是 Stream<String>而不是 Stream<List<String>>
    Stream<String> stream = Arrays.asList("123", "234").stream();
}
```

C、获取数组中数据的流，可以用 Arrays.stream(Array)获取数组里面内容的流。
见下面案列：

```
private void streamContentTest(){
    //获取到的是集合里面数据的流，比如此处集合中的数据是字符串，此处获取到的就是 Stream<String>而不是 Stream<List<String>>
    Stream<String> stream = Arrays.stream(new String[]{"123","234"});
}
```

(2)、流的分类（具体用法见第六章）

中间流（返回 Stream）： map (mapToInt, flatMap 等)、 filter、 distinct、 sorted、 peek、 limit、 skip、 parallel、 sequential、 unordered。

终端流（返回你所期望的类型）： forEach、 forEachOrdered、 toArray、 reduce、 collect、 min、 max、 count、 anyMatch、 allMatch、 noneMatch、 findFirst、 findAny、 iterator 。

短路流（返回 Stream）：anyMatch、allMatch、noneMatch、findFirst、findAny、limit。
可以简单的理解为：
由 Stream.of(T...t)产生河里面的流水，流到河中间我们可以对河里面的流水进行一些操作，比如过滤杂质（filter），加点盐让它变成盐水（map），这些就是中间流。我们选择其中的一部分（findAny），将选择的部分用瓶装起来（collect）。这两步分别扮演了短路流和终端流的角色。可以看到整个过程就像流水一样，只能从前到后，不能从后到前。

7、Optional
Optional 是 Java8 新加入的一个容器，这个容器只存 1 个或 0 个元素，它用于防止出现 NullpointException。

六、 操作符

每一个流都有中间流，短路流和终端流，俗称操作符（reactor 中的概念），这些操作符由 java 为我们提供，每一个操作符对应不同的功能，java 中的 lambda 表达式就是对各种操作符的使用。

1、map 操作符

原型：

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

功能：见名知意，“转换”，将类型为 T 的流转换为 R 的流。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 Function 需要一个参数类型为 T 的数据，返回的是一个 R 类型的数据。

案列：

```
//转换案例
//将链表或数组中的每个元素进行某个批量操作，所有元素合在一起返回数组或链表
public void mapOfStream() {

      //初始化数组
        String[] strings = new String[]{"1", "2", "3", "4", "5"};
      //将 String 数组转换为 Integer 链表
    List<Integer> integers = Stream.of(strings).map(string ->
Integer.parseInt(string)).collect(Collectors.toList());

        for (Integer integer : integers) {
            System.out.println(integer);
         }

    //将数组中的每个数据添加一个加号
    String[] array = Stream.of(strings).map(s -> {
        return s + "+";
    }).toArray(String[]::new);

        for (String s : array) {
           System.out.println(s);
          }

}
```

2、filter 操作符

原型：

```
Stream<T> filter(Predicate<? super T> predicate);
```

功能：见名知意，“过滤”，过滤掉不符合条件的数据。注意断定原型是留下满足条件的数据。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 Predicate 需要一个参数为 T 类型的数据，返回一个布尔类型数据。

案列：

```
/**
 * 过滤案例
 * 要求匿名函数返回值为 boolean
 */
```

```
public void filterOfStream() {

        Integer[] integers = new Integer[]{1, 2, 3, 4, 5};
      //取出数组中的偶数,filter 中的匿名函数返回值为 boolean
    Integer[] array = Stream.of(integers).filter(a -> a % 2 ==
0).toArray(Integer[]::new);

         for (Integer integer : array) {
             System.out.println(integer);
           }

    List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
    //filter 方法会去调用 Predicate 中的 test 方法
    //filter((a) -> a % 2 == 0)和 filter(a -> a > 0) 箭头前面是参数，后面的是返
回值，只有一个参数时可以省略参数的括号
    List<Integer> collect = integerList.stream().filter((a) -> a % 2 == 0).filter(a
-> a > 0).collect(Collectors.toList());
    for (Integer integer : collect) {
        System.out.println(integer);
    }
}
```

3、distinct 操作符

原型：

```
Stream<T> distinct();
```

功能：见名知意，“去重”，对流进行去重。

分析：很明显可以看出，此操作符不需要任何参数，也不返回任何值。故而直接使用即可。

案例：

```
//排序案例
public void sortedOfStream() {

    //先去重，由大到小排序
    List<Integer> list = new ArrayList<Integer>() {
        {
            add(2);
            add(1);
            add(5);
            add(6);
            add(7);
            add(1);
        }
    };
```

```
    List<Integer> collect = list.stream().distinct().sorted((a, b) -> a > b ? -1 :
1).collect(Collectors.toList());
    for (Integer integer : collect) {
        System.out.println(integer);
    }

}
```

4、peek 操作符

原型：

```
Stream<T> peek(Consumer<? super T> action);
```

功能：见名知意，“偷看，偷窥”。在流处理过程中进行一些其他操作，流本身不进行任何改变。多用于打印日志和对可访问域变量进行操作。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 Consumer 需要一个参数为 T 类型的数据，不返回任何值。

案列：

```
//peek 案例
public void peekOfStream() {

    //peek 不是终结操作，不会将流 consume 掉
    List<String> collect = Stream.of("one", "two", "three", "four")
            .peek(e -> System.out.println("Peeked value: " + e))
            .map(String::toUpperCase)
            .peek(e -> System.out.println("Mapped value: " + e))
            .collect(Collectors.toList());
    for (String s : collect) {
        System.out.println(s);
    }
}
```

5、foreach 操作符

原型：

```
void forEach(Consumer<? super T> action);
```

功能：见名知意，“遍历”，对流中数据进行遍历操作。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 Consumer 需要一个参数为 T 类型的数据，不返回任何值。

案列：

```
//遍历案例，并行遍历，不能保证遍历顺序
public void forEachOfStream() {
    List<String> list = Arrays.asList("hello world", "hello java", "hello c#",
"hello git", "hello php");

    System.out.println("开始遍历输出链表数据...");
    list.stream().forEach(s -> {
```

```
        System.out.println(s);
    });
    System.out.println("遍历输出链表数据完成...");


    Map<String, Integer> map = new ConcurrentHashMap<>();
    map.put("fang", 100);
    map.put("xiao", 200);
    map.put("yong", 300);
    map.forEach((key, value) -> {
        System.out.println("key:  " + key + " value:  " + value);
    });

}

//按顺序遍历案例，属于串行遍历，效率不如 foreach 快，foreach 是并行遍历，但是不能保证遍历顺序
public void forEachOrderedOfStream() {
    List<Integer> list = Arrays.asList(1, 2, 3, 54, 43, 1, 4324, 0, 34, 33);
    System.out.println("开始遍历输出链表数据...");
    list.stream().forEachOrdered(a -> System.out.println(a));
    System.out.println("遍历输出链表数据完成...");

    Map<String, Integer> map = new HashMap<>();
    map.put("fang", 100);
    map.put("xiao", 200);
    map.put("yong", 300);
    map.forEach((key, value) -> {
        System.out.println("key:  " + key + " value:  " + value);
    });
}
```

6、reduce 操作符

原型：

```
Optional<T> reduce(BinaryOperator<T> accumulator);
```

功能：见名知意，“减少”，将流中的数据减少，这里的减少不是传统意义上的直接丢掉某些数据，而是对数据进行某些操作后，流的数据个数减少。比如，流中有 a,b,c 三个整形数据，我对流中的数据进行求和操作，返回求和后的数据，那此处流中的数据个数由 3 个变为 1 个，这就是减少操作。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 BinaryOperator 需要两个同类型 T 的数据，返回一个同 T 类型的数据。

案列：

```
//减少案例
public void reduceOfStream() {
```

```
    //reduce()函数接收两个参数{上一次计算结果,stream 中的项}，返回 optional 接口
    // 三个参数时，{初始值，{上一次计算结果,stream 中的项}}，返回 reduce 中指定的
类型

    List<Integer> list = Arrays.asList(21, 43, 234, 5, 7, 8, 1, 0);

    //两个参数实现累加操作
    Optional<Integer> optional = list.stream().filter(a -> a !=
0).distinct().reduce((result, item) -> {
        //result 初始值为 stream 中的第一个数据，item 初始值为第二数据
        return result + item;
    });
    System.out.println("用两个参数的方法计算数组累加的和为：" + optional.get());

    //三个参数实现累加操作
    Integer sum = list.stream().filter(a -> a != 0).distinct().reduce(0, (result,
item) -> {
    //result 初始值为前面个指定的参数，即 0，item 初始值为 stream 中的第一个参数
        return result + item;
    });
    System.out.println("用三个参数的方法计算数组累加的和是：" + sum);

    //直接调用函数实现累加，根据业务逻辑来调用自定义的函数
    Optional<Integer> integerOptional = list.stream().filter(a -> a !=
0).distinct().reduce(Integer::sum);
    Integer integer = list.stream().filter(a -> a != 0).distinct().reduce(0,
Integer::sum);
    System.out.println("调用函数求累加和：" + integerOptional.get() + "--->" +
integer);
}
```

7、collect 操作

原型：

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

功能：见名知意，“收集”，将流中的数据用收集器收集起来。概念请看第五章。具体用法见第七章。

分析：由参数可见，此处需要一个收集器。收集器可以自己定义，也可以使用收集器工具类为我们提供的收集器。

案列：

```
//集合案例
public void collectOfStream() {
    String[] strings = new String[]{"123", "23", "432"};
    List<String> collect = Stream.of(strings).collect(Collectors.toList());
```

```
    collect.stream().forEachOrdered(a -> {
        System.out.println(a);
    });

    //或者，如果需要输出某些对象的属性，需要用花括号来分开写
    collect.stream().forEachOrdered(System.out::println);
}
```

8、allMatch 操作符

原型：

```
boolean allMatch(Predicate<? super T> predicate);
```

功能：见名知意，“全匹配”，查看流中的数据是否全部匹配条件，全部匹配则返回 true，否则返回 false。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 Predicate 需要一个参数为 T 类型的数据，返回一个布尔类型数据。

案列：

```
//全匹配案例
public void allMatchOfStream() {
    List list = Arrays.asList(23, "343", "45645");
    boolean allMatch = list.stream().allMatch(a -> judgeConditions(a));
    boolean anyMatch = list.stream().anyMatch(a -> judgeConditions(a));
    boolean noneMatch = list.stream().noneMatch(a -> judgeConditions(a));
    System.out.println(allMatch);
    System.out.println(anyMatch);
    System.out.println(noneMatch);
}

//匹配条件
private boolean judgeConditions(Object o) {
    return o instanceof String ? true : false;
}
```

9、limit 操作符

原型：

```
Stream<T> limit(long maxSize);
```

功能：见名知意，“限制”，限制流中数据的最大条数。

分析：很显然，此处仅仅需要一个整形参数，即最大数据的条数。

案列：

```
//限制案例
public void limitOfStream() {

    //先去重，再排序，最后限制条数
    List<Integer> list = Arrays.asList(1, 4, 3, 4, 3, 6, 3, 2);
    List<Integer> collect = list.stream().distinct().sorted((a, b) -> a > b ? 1 :
```

```
-1).limit(3).collect(Collectors.toList());
    for (Integer integer : collect) {
        System.out.println(integer);
    }
}
```

10、flatMap 操作符

原型：

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

功能：见名知意，”平坦映射，平坦转换”，将多维流转换为更低级的流。如二维数组的流经过平坦转换后就转变为一维数组的流，一维数组的流经过平坦转换后就转变为了一维数组中数据的流。

分析：通过第四章 lambda 表达式原型我们可以知道，此处的 Function 需要一个参数类型为 T 的数据，返回的是一个 Stream 类型的数据。

案列：

```
/**
 * 扁平化转换案例
 * 进行某种函数转换，将二维数组转换为一维数组
 */
public void flatMapOfStream() {

    //将二维数组转换为一维数组
    String[][] strings = new String[][]{{"fang"}, {"xiao"}, {"yong"}, {"is"},
{"best"}};
    //先获取二维数组中子项的流，即一维数组，再获取一维数组中子项的流，即每个元素
    List<String> stringList = Arrays.stream(strings)
            .flatMap(a -> Arrays.stream(a)).collect(Collectors.toList()); //将二
维数组转换为一维数组

    stringList.stream().forEachOrdered(a -> {
        System.out.println(a);
    });

    //将一维数组转换为二维数组，遍历方法省略
    List<String> list = Arrays.asList("fang", "xiao", "yong");
    List<String[][]> collect = list.stream().map(a -> new
String[][]{{a}}).collect(Collectors.toList());

}
```

以上是常用的操作符，部分比较不常用的操作符，如有兴趣可以自行学习，大致分析思路参考案列部分即可。

七、 集合操作

集合是 java 中最常用的数据结构，操作也比较复杂，是应用开发过程中阻碍进度的罪魁祸首，某种意义上来说，集合的操作占据整个业务代码的 80%或者以上。Java8 中的 Collectors 专用与集合操作的优化，做到了化繁为简。极大加快应用开发进度。

上文中我们介绍到各种操作符，都是针对流的操作，那我们怎样把操作符处理过的流转换为我们需要的数据结构呢？此时，就需要 Collector 了。Collector（收集器）是专用于收集流的 api。但是自定义收集器比较复杂，java8 已经帮我们定义了大量的收集器 Collectors。直接引用即可。

1、普通集合收集器（list，set）

原型：

```
//list 收集器原型
public static <T> Collector<T, ?, List<T>> toList();
//set 收集器原型
public static <T> Collector<T, ?, Set<T>> toSet();
```

功能：用 list 或者 set 集合将处理后的流数据收集起来。

分析：很显然，两个收集器都是无参函数，直接使用即可。

案列：

```
//普通集合收集器
private void normalCollector(){
    //初始化链表
    List<String> strings = Arrays.asList("fang", "xiao", "yong", "go", "!");

    //过滤掉叹号的字符串，并用 List 进行收集
    List<String> list = strings.stream().filter(a ->
a.equals("!")).collect(Collectors.toList());

    //将链表进行反序操作，然后用 Set 进行收集
    Set<String> set =
strings.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toSet());
}
```

2、map 收集器

原型：

```
public static <T, K, U>
Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
                                Function<? super T, ? extends U> valueMapper,
                                BinaryOperator<U> mergeFunction);
```

功能：用 map 将处理后的流收集起来，key 和 value 自定义。

分析：可以看出该收集器需要三个参数，由参数名可知第一个是键，第二个是 value，第三个是合并器，用于当 key 相同时的处理方案。再结合第 4 章 lambda 表达原型可知，keyMapper 和 valueMapper 是一个 Function，需要一个 T 类型的

数据，返回值分别为 K 类型和 U 类型的数据。这里的 K 类型数据和 U 类型的数据没有太大的含义，可以简单的理解为就是需要一个返回值。MergeFunction 是一个二元操作者，故而需要两个同类型为 U 的参数，返回一个 U 类型的数据。
案列：

```
//map 收集器
private void mapCollector(){
    //初始化列表
    List<Pojo> pojos = Arrays.asList(new Pojo("123", "fang"), new Pojo("456",
"xiao"), new Pojo("123", "yong"));

    //转换为 map，key 为 id,value 为对象，如果出现两个 key 相同的情况，以第一个为准
    Map<String, Pojo> id2MapOne = pojos.stream().collect(Collectors.toMap(a ->
a.getId(), b -> b,(c,d)->c));

    //方式二
    Map<String, Pojo> id2MapTwo =
pojos.stream().collect(Collectors.toMap(Pojo::getId, a -> a, (b, c) -> b));
}
```

3、分组收集器
原型：

```
public static <T, K, A, D>
Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
                                      Collector<? super T, A, D> downstream);
```

功能：对处理后的流进行分组操作，然后用 map 进行收集。
分析：由参数名可知，第一个参数 classifier 是分组标志，第二个是下级流的收集器。再结合第四章 lambda 原型可知，classifier 需要一个 T 类型的数据，返回一个 K 类型的数据，downstream 是分组数据的收集器。
案列：

```
//分组收集器
private void groupCollector(){
    //初始化列表
    List<Pojo> pojos = Arrays.asList(new Pojo("123", "fang"), new Pojo("456",
"xiao"), new Pojo("123", "yong"));

    //方式一，默认分组收集器为 List
    Map<String, List<Pojo>> groupByIdOne =
pojos.stream().collect(Collectors.groupingBy(Pojo::getId));

    //方式二，自定义分组收集器
    Map<String, Set<Pojo>> groupByIdTwo =
pojos.stream().collect(Collectors.groupingBy(a -> a.getId(),
Collectors.toSet()));
```

```
    //方式三，自定义分组收集器，且用指定的 map 进行分组，可以保留原 map 中的数据
    Map<String, List<Pojo>> groupByThree =
pojos.stream().collect(Collectors.groupingBy(Pojo::getId, Collections::emptyMap,
Collectors.toList()));

}
```

4、分区收集器

原型：

```
public static <T, D, A>
Collector<T, ?, Map<Boolean, D>> partitioningBy(Predicate<? super T> predicate,
                                                Collector<? super T, A, D>
downstream) ;
```

功能：对流中的数据进行分区操作，分区结果仅仅有两种结果，一种是满足条件的 true 分区，不满足条件的 false 分区。

分析：由参数名可知，predicate 为断言，第二个参数是自定义下级流收集器。再结合第四章 lambda 表达式原型可知 predicate 需要一个 T 类型的数据，返回一个布尔类型的数据。downstream 为下级流收集器。

案列：

```
//分区收集器
private void partitionCollector(){
    //初始化列表
    List<Pojo> pojos = Arrays.asList(new Pojo("123", "fang"), new Pojo("456",
"xiao"), new Pojo("123", "yong"));

   //用 id 进行分区，满足条件的在一个 true 区，不满足条件的在 false 区
    Map<Boolean, List<Pojo>> partitionByIdOne =
pojos.stream().collect(Collectors.partitioningBy(a -> a.getId().equals("123")));

  //用 id 进行分区，并指定下级流收集器
   Map<Boolean, Set<Pojo>> partitonByIdTwo =
pojos.stream().collect(Collectors.partitioningBy(a -> a.getId().equals("123"),
Collectors.toSet()));
}
```

5、数据统计收集器

原型：

```
public static <T>
Collector<T, ?, IntSummaryStatistics> summarizingInt(ToIntFunction<? super T>
mapper);
```

功能：对数据进行统计操作，然后返回数据统计数据 IntSummaryStatistics，统计数据中包含总和，平均值，最大值，最小值，个数等数据。

分析：由第四章 lambda 表达式原型，可以知道，mapper 需要一个 T 类型的数据，返回一个整形数据。

案列：

```
//数据统计收集器
private void summingCollector(){
    //数据统计收集器
    IntSummaryStatistics intSummaryStatistics = Arrays.asList(123, 123, 678, 890,
200).stream().collect(Collectors.summarizingInt(a -> a));
    //取得平均值
    double average = intSummaryStatistics.getAverage();
    //取得数据个数
    long count = intSummaryStatistics.getCount();
    //取得最大数
    int max = intSummaryStatistics.getMax();
    //取得最小数
    int min = intSummaryStatistics.getMin();
    //取得总和
    long sum = intSummaryStatistics.getSum();
}
```

以上是常用的收集器，部分比较不常用的收集器，如有兴趣可以自行学习，大致分析思路参考案列部分即可。

八、 冒号表达式

1、冒号的作用

A、静态方法引用(Reference to a static method)

语法：ContainingClass::staticMethodName

例如：Person::getAge

B、对象的实例方法引用(Reference to an instance method of a particular obj)

语法：containingObject::instanceMethodName

例如：System.out::println

C、特定类型的任意对象实例的方法(Reference to an instance method of an arbitrary object of a particular type)

语法：(ContainingType::methodName)

例如：String::compareToIgnoreCase

D、类构造器引用语法(Reference to a constructor)

语法：ClassName::new

例如：ArrayList::new

2、使用技巧

A、双冒号表达式返回的是一个函数式接口对象（@FunctionalInterface 实例）

B、大胆的使用冒号表达式，不要怕出错。Idea 现在已经很智能了，如果编译器推断不出来冒号表达式，再换为普通的 lambda 表达式即可。

九、 Optional 容器

Optional 是 Java8 新加入的一个容器，这个容器只存 1 个或 0 个元素，它用于防止出现 NullpointException，它提供如下方法：

Boolean isPresent() 判断容器中是否有值。

Boolean ifPresent(Consume lambda) 容器若不为空则执行括号中的 Lambda 表达式。

T get() 获取容器中的元素，若容器为空则抛出 NoSuchElement 异常。

T orElse(T other) 获取容器中的元素，若容器为空则返回括号中的默认值。

其余常用方法请参见案列：

```
//optional 容器案列
public void option() throws Exception {

    //将抛空指针异常
    Optional<Object> optionalOne = Optional.of(null);

    //将返回 nul，不会抛异常，此方法一般用的比较多
    Optional<Object> optionalTwo = Optional.ofNullable(null);

    Optional<List<String>> optionalList =
Optional.ofNullable(Arrays.asList("123", "234", "345"));

    //如果容器不为空
    if (optionalList.isPresent()) {
        //取得容器中的值
        List<String> list = optionalList.get();
        //也可以在容器有值时进行一些其他操作,有值时以同步的方式输出链表中的数据
        optionalList.ifPresent(a->a.stream().forEach(System.out::println));
    } else {
        //可以以传统形式设置容器为空时的默认值
        List<String> customList = optionalList.orElse(new ArrayList<>());

        //也可以以 lambda 表达式设置当容器为空时返回一个默认值
        List<String> lambdaList = optionalList.orElseGet(ArrayList::new);

        //亦可以抛出异常
        List<String> exceptionOrList = optionalList.orElseThrow(Exception::new);
    }
}
```

十、 其他事项

本文档所有代码已上传至 **github**：https://github.com/JOETION/phoenix-demo

参考网址：

【1】Reactive Programming vs. Reactive stream
https://www.tuicool.com/articles/UZfMB3M

【2】 Reactive Programming with JDK 9 Flow API
https://community.oracle.com/docs/DOC-1006738

【3】Netty（RPC 高性能之道）原理剖析
https://blog.csdn.net/zhiguozhu/article/details/50517551

【4】What Are Reactive Streams in Java
https://dzone.com/articles/what-are-reactive-streams-in-java

【5】Reactor 实例解析
http://www.open-open.com/lib/view/open1482286087274.html

【6】使用 Reactor 进行反应式编程
https://www.ibm.com/developerworks/cn/java/j-cn-with-reactor-response-encode/index.html

【7】Java ProjectReactor 框架之 Flux 篇
https://www.jianshu.com/p/8cb66e912641

【8】聊聊 Spring Boot 2.0 的 WebFlux
https://www.bysocket.com/?p=1987

【9】使用 Spring 5 的 WebFlux 开发反应式 Web 应用
https://www.ibm.com/developerworks/cn/java/spring5-webflux-reactive/index.html