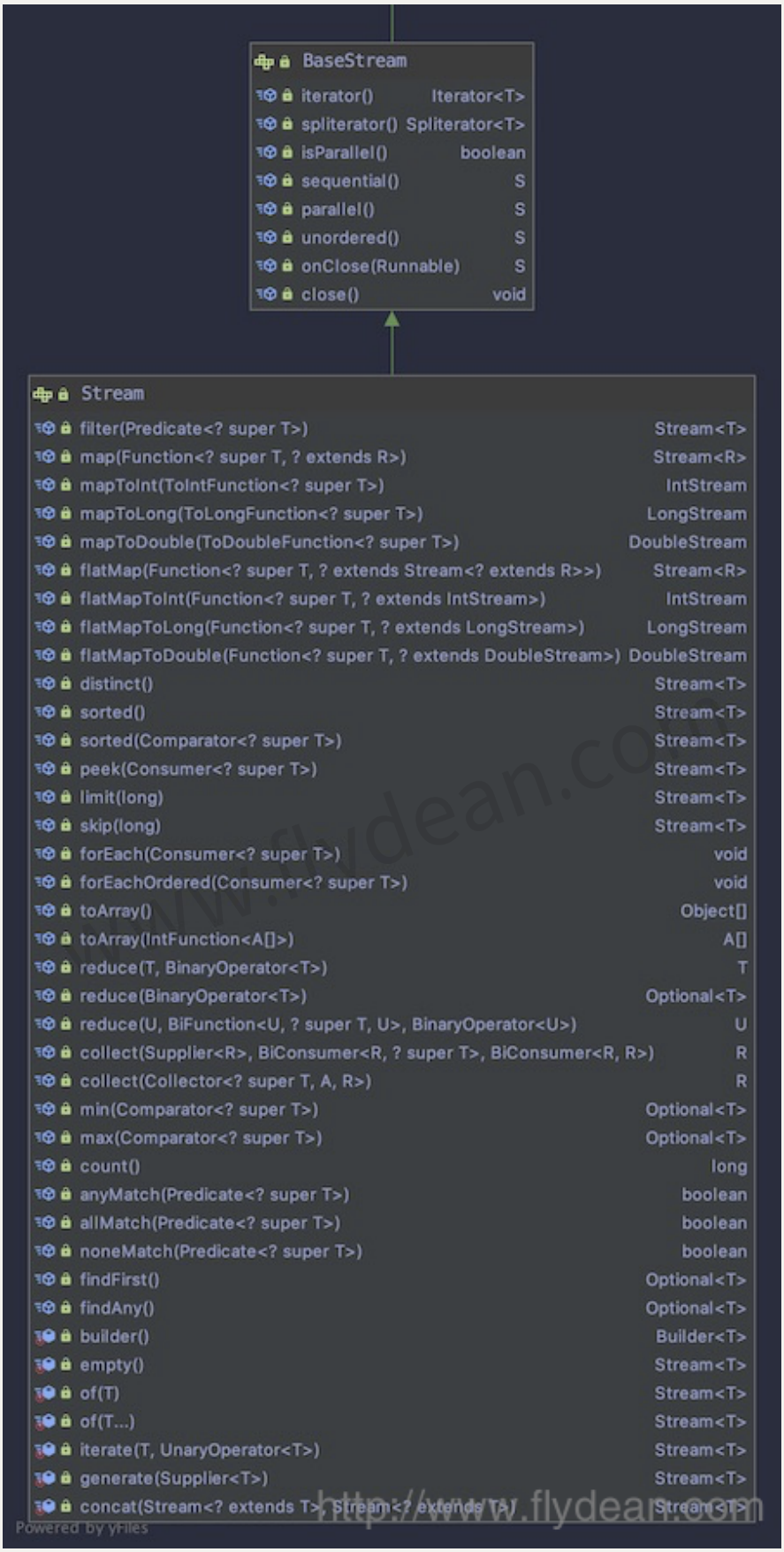
Stream和Lambda表达式最佳实践

1. Streams简介

今天要讲的Stream指的是java.util.stream包中的诸多类。Stream可以方便的将之前的结合类以转换为Stream并以流式方式进行处理，大大的简化了我们的编程，Stream包中，最核心的就是interface Stream


BaseStream
iterator() Iterator<T>
spliterator() Spliterator<T>
isParallel() boolean
sequential() S
parallel() S
unordered() S
onClose(Runnable) S
close() void
Stream
filter(Predicate<? super T>) Stream<T>
map(Function<? super T, ? extends R>) Stream<R>
mapToInt(ToIntFunction<? super T>) IntStream
mapToLong(ToLongFunction<? super T>) LongStream
mapToDouble(ToDoubleFunction<? super T>) DoubleStream
flatMap(Function<? super T, ? extends Stream<? extends R>>) Stream<R>
flatMapToInt(Function<? super T, ? extends IntStream>) IntStream
flatMapToLong(Function<? super T, ? extends LongStream>) LongStream
flatMapToDouble(Function<? super T, ? extends DoubleStream>) DoubleStream
distinct() Stream<T>
sorted() Stream<T>
sorted(Comparator<? super T>) Stream<T>
peek(Consumer<? super T>) Stream<T>
limit(long) Stream<T>
skip(long) Stream<T>
forEach(Consumer<? super T>) void
forEachOrdered(Consumer<? super T>) void
toArray() Object[]
toArray(IntFunction<A[]>) A[]
reduce(T, BinaryOperator<T>) T
reduce(BinaryOperator<T>) Optional<T>
reduce(U, BiFunction<U, ? super T, U>, BinaryOperator<U>) U
collect(Supplier<R>, BiConsumer<R, ? super T>, BiConsumer<R, R>) R
collect(Collector<? super T, A, R>) R
min(Comparator<? super T>) Optional<T>
max(Comparator<? super T>) Optional<T>
count() long
anyMatch(Predicate<? super T>) boolean
allMatch(Predicate<? super T>) boolean
noneMatch(Predicate<? super T>) boolean
findFirst() Optional<T>
findAny() Optional<T>
builder() Builder<T>
empty() Stream<T>
of(T) Stream<T>
of(T...) Stream<T>
iterate(T, UnaryOperator<T>) Stream<T>
generate(Supplier<T>) Stream<T>
concat(Stream<? extends T>, Stream<? extends T>) Stream<T>
Powered by yFiles
http://www.flydean.com

从上面的图中我们可以看到Stream继承自BaseStream。Stream中定义了很多非常实用的方法，比如filter，map，flatmap,forEach,reduce,collect等等。接下来我们将会逐一讲解。

1.1 创建Stream

Stream的创建有很多方式，java引入Stream之后所有的集合类都添加了一个stream()方法，通过这个方法可以直接得到其对应的Stream。也可以通过Stream.of方法来创建：

```
//Stream Creation
        String[] arr = new String[]{"a", "b", "c"};
        Stream<String> stream = Arrays.stream(arr);
        stream = Stream.of("a", "b", "c");
```

1.2 Streams多线程

如果我们想使用多线程来处理集合类的数据，Stream提供了非常方便的多线程方法parallelStream()：

```
//Multi-threading
        List<String> list =new ArrayList();
        list.add("aaa");
        list.add("bbb");
        list.add("abc");
        list.add("ccc");
        list.add("ddd");
        list.parallelStream().forEach(element ->
doPrint(element));
```


1.3 Stream的基本操作

Stream的操作可以分为两类，一类是中间操作，中间操作返回Stream，因此可以级联调用。 另一类是终止操作，这类操作会返回Stream定义的类型。

```
//Operations
        long count = list.stream().distinct().count();
```

上面的例子中，distinct()返回一个Stream，所以可以级联操作，最后的count()是一个终止操作，返回最后的值。

Matching

Stream提供了anyMatch(), allMatch(), noneMatch()这三种match方式，我们看下怎么使用：

```
//Matching
        boolean isValid = list.stream().anyMatch(element
-> element.contains("h"));
        boolean isValidOne =
list.stream().allMatch(element ->
element.contains("h"));
        boolean isValidTwo =
list.stream().noneMatch(element ->
element.contains("h"));
```

Filtering

filter() 方法允许我们对Stream中的数据进行过滤，从而得到我们需要的：

```
Stream<String> filterStream =
list.stream().filter(element -> element.contains("d"));
```

上面的例子中我们从list中选出了包含“d”字母的String。

Mapping

map就是对Stream中的值进行再加工，然后将加工过后的值作为新的Stream返回。

```
//Mapping
        Stream<String> mappingStream =
list.stream().map(element -> convertElement(element));

    private static String convertElement(String element)
{
        return "element"+"abc";
    }
```

上的例子中我们把list中的每个值都加上了“abc”然后返回一个新的Stream。

FlatMap

flatMap和Map很类似，但是他们两个又有不同，看名字我们可以看到flatMap意思是打平之后再做Map。

怎么理解呢？

假如我们有一个CustBook类：

```
@Data
public class CustBook {

    List<String> bookName;
}
```

CustBook定义了一个bookName字段。

先看一下Map返回的结果：

```
List<CustBook> users = new ArrayList<>();
        users.add(new CustBook());
Stream<Stream<String>> userStreamMap
                = users.stream().map(user ->
user.getBookName().stream());
```

在上面的代码中，map将每一个user都转换成了stream，所以最后的结果是返回Stream的Stream。

如果我们只想返回String，则可以使用FlatMap：

```
List<CustBook> users = new ArrayList<>();
        users.add(new CustBook());
        Stream<String> userStream
            = users.stream().flatMap(user ->
user.getBookName().stream());
```


简单点讲FlatMap就是将层级关系铺平重来。

Reduction

使用reduce() 方法可以方便的对集合的数据进行运算，reduce()接收两个参数，第一个是开始值，后面是一个函数表示累计。

```
//Reduction
        List<Integer> integers = Arrays.asList(1, 1, 1);
        Integer reduced = integers.stream().reduce(100,
(a, b) -> a + b);
```

上面的例子我们定义了3个1的list，然后调用reduce(100, (a, b) -> a + b)方法，最后的结果是103.

Collecting

collect()方法可以方便的将Stream再次转换为集合类，方便处理和展示：

```
List<String> resultList
                = list.stream().map(element ->
element.toUpperCase()).collect(Collectors.toList());
```

2. functional interface的分类和使用

java 8引入了lambda表达式，lambda表达式实际上表示的就是一个匿名的function。

在java 8之前，如果需要使用到匿名function需要new一个类的实现，但是有了lambda表达式之后，一切都变的非常简介。

我们看一个之前讲线程池的时候的一个例子：

```
//ExecutorService using class
        ExecutorService executorService =
Executors.newSingleThreadExecutor();
        executorService.submit(new Runnable() {
            @Override
            public void run() {
            log.info("new runnable");
            }
        });
```


executorService.submit需要接收一个Runnable类，上面的例子中我们new了一个Runnable类，并实现了它的run（）方法。

上面的例子如果用lambda表达式来重写，则如下所示：

```
//ExecutorService using lambda
        executorService.submit(()->log.info("new
runnable"));
```

看起是不是很简单，使用lambda表达式就可以省略匿名类的构造，并且可读性更强。

那么是不是所有的匿名类都可以用lambda表达式来重构呢？也不是。

我们看下Runnable类有什么特点：

```
@FunctionalInterface
public interface Runnable
```

Runnable类上面有一个@FunctionalInterface注解。这个注解就是我们今天要讲到的Functional Interface。

2.1 Functional Interface

Functional Interface是指带有 @FunctionalInterface 注解的interface。它的特点是其中只有一个子类必须要实现的abstract方法。如果abstract方法前面带有default关键字，则不做计算。

其实这个也很好理解，因为Functional Interface改写成为lambda表达式之后，并没有指定实现的哪个方法，如果有多个方法需要实现的话，就会有问题。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

Functional Interface一般都在java.util.function包中。

根据要实现的方法参数和返回值的不同，Functional Interface可以分为很多种，下面我们分别来介绍。

2.2 Function：一个参数一个返回值

Function接口定义了一个方法，接收一个参数，返回一个参数。

```
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
```

一般我们在对集合类进行处理的时候，会用到Function。

```
Map<String, Integer> nameMap = new HashMap<>();
        Integer value = nameMap.computeIfAbsent("name",
s -> s.length());
```

上面的例子中我们调用了map的computeIfAbsent方法，传入一个Function。

上面的例子还可以改写成更短的：

```
Integer value1 = nameMap.computeIfAbsent("name",
String::length);
```

Function没有指明参数和返回值的类型，如果需要传入特定的参数，则可以使用IntFunction, LongFunction, DoubleFunction：

```
@FunctionalInterface
public interface IntFunction<R> {

    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    R apply(int value);
}
```


如果需要返回特定的参数，则可以使用ToIntFunction, ToLongFunction, ToDoubleFunction：

```
@FunctionalInterface
public interface ToDoubleFunction<T> {

    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    double applyAsDouble(T value);
}
```

如果要同时指定参数和返回值，则可以使用DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction：

```
@FunctionalInterface
public interface LongToIntFunction {

    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    int applyAsInt(long value);
}
```

2.3 BiFunction：接收两个参数，一个返回值

如果需要接受两个参数，一个返回值的话，可以使用BiFunction：BiFunction, ToDoubleBiFunction, ToIntBiFunction, ToLongBiFunction等。

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    /**
     * Applies this function to the given arguments.
     *
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    R apply(T t, U u);
```


我们看一个BiFunction的例子：

```
//BiFunction
        Map<String, Integer> salaries = new HashMap<>();
        salaries.put("alice", 100);
        salaries.put("jack", 200);
        salaries.put("mark", 300);

        salaries.replaceAll((name, oldValue) ->
                name.equals("alice") ? oldValue :
oldValue + 200);
```

2.4 Supplier：无参的Function

如果什么参数都不需要，则可以使用Supplier：

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

2.5 Consumer：接收一个参数，不返回值

Consumer接收一个参数，但是不返回任何值，我们看下Consumer的定义：

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
```


看一个Consumer的具体应用：

```
//Consumer
        nameMap.forEach((name, age) ->
System.out.println(name + " is " + age + " years old"));
```

2.6 Predicate：接收一个参数，返回boolean

Predicate接收一个参数，返回boolean值：

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument
matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
```

如果用在集合类的过滤上面那是极好的：

```
//Predicate
        List<String> names = Arrays.asList("A", "B",
"C", "D", "E");
        List<String> namesWithA = names.stream()
                .filter(name -> name.startsWith("A"))
                .collect(Collectors.toList());
```

2.7 Operator：接收和返回同样的类型

Operator接收和返回同样的类型，有很多种Operator：UnaryOperator BinaryOperator，DoubleUnaryOperator, IntUnaryOperator, LongUnaryOperator, DoubleBinaryOperator, IntBinaryOperator, LongBinaryOperator等。

```
@FunctionalInterface
public interface IntUnaryOperator {

    /**
     * Applies this operator to the given operand.
     *
     * @param operand the operand
     * @return the operator result
     */
    int applyAsInt(int operand);
```

我们看一个BinaryOperator的例子：

```
 //Operator
        List<Integer> values = Arrays.asList(1, 2, 3, 4,
5);
        int sum = values.stream()
                .reduce(0, (i1, i2) -> i1 + i2);
```

3. Lambda表达式最佳实践

Lambda表达式java 8引入的函数式编程框架。之前的文章中我们也讲过Lambda表达式的基本用法。

本文将会在之前的文章基础上更加详细的讲解Lambda表达式在实际应用中的最佳实践经验。

3.1 优先使用标准Functional接口

之前的文章我们讲到了，java在java.util.function包中定义了很多Function接口。基本上涵盖了我们能够想到的各种类型。

假如我们自定义了下面的Functional interface：

```
@FunctionalInterface
public interface Usage {
    String method(String string);
}
```


然后我们需要在一个test方法中传入该interface：

```
public String test(String string, Usage usage) {
    return usage.method(string);
}
```

上面我们定义的函数接口需要实现method方法，接收一个String，返回一个String。这样我们完全可以使用Function来代替：

```
public String test(String string, Function<String,
String> fn) {
    return fn.apply(string);
}
```

使用标准接口的好处就是，不要重复造轮子。

3.2 使用@FunctionalInterface注解

虽然@FunctionalInterface不是必须的，不使用@FunctionalInterface也可以定义一个Functional Interface。

但是使用@FunctionalInterface可以在违背Functional Interface定义的时候报警。

如果是在维护一个大型项目中，加上@FunctionalInterface注解可以清楚的让其他人了解这个类的作用。

从而使代码更加规范和更加可用。

所以我们需要这样定义：

```
@FunctionalInterface
public interface Usage {
    String method(String string);
}
```

而不是：

```
public interface Usage {
    String method(String string);
}
```


3.3 在Functional Interfaces中不要滥用Default Methods

Functional Interface是指只有一个未实现的抽象方法的接口。

如果该Interface中有多个方法，则可以使用default关键字为其提供一个默认的实现。

但是我们知道Interface是可以多继承的，一个class可以实现多个Interface。 如果多个Interface中定义了相同的default方法，则会报错。

通常来说default关键字一般用在升级项目中，避免代码报错。

3.4 使用Lambda 表达式来实例化Functional Interface

还是上面的例子：

```
@FunctionalInterface
public interface Usage {
    String method(String string);
}
```

要实例化Usage，我们可以使用new关键词：

```
Usage usage = new Usage() {
    @Override
    public String method(String string) {
        return string;
    }
};
```

但是最好的办法就是用lambda表达式：

```
Usage usage = parameter -> parameter;
```

3.5 不要重写Functional Interface作为参数的方法

怎么理解呢？ 我们看下面两个方法：

```
public class ProcessorImpl implements Processor {
    @Override
    public String process(Callable<String> c) throws
Exception {
        // implementation details
    }

    @Override
    public String process(Supplier<String> s) {
        // implementation details
    }
}
```

两个方法的方法名是一样的，只有传入的参数不同。但是两个参数都是Functional Interface，都可以用同样的lambda表达式来表示。

在调用的时候：

```
String result = processor.process(() -> "test");
```

因为区别不了到底调用的哪个方法，则会报错。

最好的办法就是将两个方法的名字修改为不同的。

3.6 Lambda表达式和内部类是不同的

虽然我们之前讲到使用lambda表达式可以替换内部类。但是两者的作用域范围是不同的。

在内部类中，会创建一个新的作用域范围，在这个作用域范围之内，你可以定义新的变量，并且可以用this引用它。

但是在Lambda表达式中，并没有定义新的作用域范围，如果在Lambda表达式中使用this，则指向的是外部类。

我们举个例子：

```
private String value = "Outer scope value";

public String scopeExperiment() {
    Usage usage = new Usage() {
        String value = "Inner class value";

        @Override
        public String method(String string) {
            return this.value;
        }
    };
    String result = usage.method("");

    Usage usageLambda = parameter -> {
        String value = "Lambda value";
        return this.value;
    };
    String resultLambda = usageLambda.method("");

    return "Results: result = " + result +
      ", resultLambda = " + resultLambda;
}
```

上面的例子将会输出“Results: result = Inner class value, resultLambda = Outer scope value”

3.7 Lambda Expression尽可能简洁

通常来说一行代码即可。如果你有非常多的逻辑，可以将这些逻辑封装成一个方法，在lambda表达式中调用该方法即可。

因为lambda表达式说到底还是一个表达式，表达式当然越短越好。

java通过类型推断来判断传入的参数类型，所以我们在lambda表达式的参数中尽量不传参数类型，像下面这样：

```
(a, b) -> a.toLowerCase() + b.toLowerCase();
```

而不是：

```
(String a, String b) -> a.toLowerCase() +
b.toLowerCase();
```

如果只有一个参数的时候，不需要带括号：

```
a -> a.toLowerCase();
```

而不是：

```
(a) -> a.toLowerCase();
```

返回值不需要带return:

```
a -> a.toLowerCase();
```

而不是：

```
a -> {return a.toLowerCase()};
```


3.8 使用方法引用

为了让lambda表达式更加简洁，在可以使用方法引用的时候，我们可以使用方法引用：

```
a -> a.toLowerCase();
```

可以被替换为：

```
String::toLowerCase;
```

3.9 Effectively Final 变量

如果在lambda表达式中引用了non-final变量，则会报错。

effectively final是什么意思呢？这个是一个近似final的意思。只要一个变量只被赋值一次，那么编译器将会把这个变量看作是effectively final的。

```
    String localVariable = "Local";
    Usage usage = parameter -> {
         localVariable = parameter;
        return localVariable;
    };
```

上面的例子中localVariable被赋值了两次，从而不是一个Effectively Final 变量，会编译报错。

为什么要这样设置呢？因为lambda表达式通常会用在并行计算中，当有多个线程同时访问变量的时候Effectively Final 变量可以防止不可以预料的修改。

4. stream表达式中实现if/else逻辑

在Stream处理中，我们通常会遇到if/else的判断情况，对于这样的问题我们怎么处理呢？

还记得我们在上一篇文章lambda最佳实践中提到，lambda表达式应该越简洁越好，不要在其中写臃肿的业务逻辑。

接下来我们看一个具体的例子。

4.1 传统写法

假如我们有一个1 to 10的list，我们想要分别挑选出奇数和偶数出来，传统的写法，我们会这样使用：

```
    public void inForEach(){
        List<Integer> ints = Arrays.asList(1, 2, 3, 4,
5, 6, 7, 8, 9, 10);

        ints.stream()
                .forEach(i -> {
                    if (i.intValue() % 2 == 0) {
                        System.out.println("i is even");
                    } else {
                        System.out.println("i is old");
                    }
                });
    }
```

上面的例子中，我们把if/else的逻辑放到了forEach中，虽然没有任何问题，但是代码显得非常臃肿。

接下来看看怎么对其进行改写。

4.2 使用filter

我们可以把if/else的逻辑改写为两个filter：

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5, 6, 7,
8, 9, 10);

        Stream<Integer> evenIntegers = ints.stream()
                .filter(i -> i.intValue() % 2 == 0);
        Stream<Integer> oddIntegers = ints.stream()
                .filter(i -> i.intValue() % 2 != 0);
```

有了这两个filter，再在filter过后的stream中使用for each：

```
        evenIntegers.forEach(i -> System.out.println("i
is even"));
        oddIntegers.forEach(i -> System.out.println("i
is old"));
```

怎么样，代码是不是非常简洁明了。

5. 在map中使用stream

Map是java中非常常用的一个集合类型，我们通常也需要去遍历Map去获取某些值，java 8引入了Stream的概念，那么我们怎么在Map中使用Stream呢？

5.1 基本概念

Map有key，value还有表示key，value整体的Entry。

创建一个Map：

```
Map<String, String> someMap = new HashMap<>();
```

获取Map的entrySet：

```
Set<Map.Entry<String, String>> entries =
someMap.entrySet();
```

获取map的key：

```
Set<String> keySet = someMap.keySet();
```

获取map的value：

```
Collection<String> values = someMap.values();
```

上面我们可以看到有这样几个集合：Map，Set，Collection。

除了Map没有stream，其他两个都有stream方法：

```
Stream<Map.Entry<String, String>> entriesStream =
entries.stream();
        Stream<String> valuesStream = values.stream();
        Stream<String> keysStream = keySet.stream();
```

我们可以通过其他几个stream来遍历map。

5.2 使用Stream获取map的key

我们先给map添加几个值：

```
someMap.put("jack","20");
someMap.put("bill","35");
```

上面我们添加了name和age字段。

如果我们想查找age=20的key，则可以这样做：

```
Optional<String> optionalName =
someMap.entrySet().stream()
                .filter(e -> "20".equals(e.getValue()))
                .map(Map.Entry::getKey)
                .findFirst();

        log.info(optionalName.get());
```

因为返回的是Optional,如果值不存在的情况下，我们也可以处理：

```
optionalName = someMap.entrySet().stream()
                .filter(e -> "Non
ages".equals(e.getValue()))
                .map(Map.Entry::getKey).findFirst();

        log.info("{}",optionalName.isPresent());
```

上面的例子我们通过调用isPresent来判断age是否存在。

如果有多个值，我们可以这样写：

```
someMap.put("alice","20");
        List<String> listnames =
someMap.entrySet().stream()
                .filter(e -> e.getValue().equals("20"))
                .map(Map.Entry::getKey)
                .collect(Collectors.toList());

        log.info("{}",listnames);
```

上面我们调用了collect(Collectors.toList())将值转成了List。

5.3 使用stream获取map的value

上面我们获取的map的key，同样的我们也可以获取map的value：

```
List<String> listAges = someMap.entrySet().stream()
                .filter(e -> e.getKey().equals("alice"))
                .map(Map.Entry::getValue)
                .collect(Collectors.toList());

        log.info("{}",listAges);
```

上面我们匹配了key值是alice的value。

6. Stream中的操作类型和peek的使用

java 8 stream作为流式操作有两种操作类型，中间操作和终止操作。这两种有什么区别呢？

我们看一个peek的例子：

```
Stream<String> stream = Stream.of("one", "two",
"three","four");
        stream.peek(System.out::println);
```

上面的例子中，我们的本意是打印出Stream的值，但实际上没有任何输出。

为什么呢？

6.1 中间操作和终止操作

一个java 8的stream是由三部分组成的。数据源，零个或一个或多个中间操作，一个或零个终止操作。

中间操作是对数据的加工，注意，中间操作是lazy操作，并不会立马启动，需要等待终止操作才会执行。

终止操作是stream的启动操作，只有加上终止操作，stream才会真正的开始执行。

所以，问题解决了，peek是一个中间操作，所以上面的例子没有任何输出。

6.2 peek

我们看下peek的文档说明：peek主要被用在debug用途。

我们看下debug用途的使用：

```
Stream.of("one", "two", "three","four").filter(e ->
e.length() > 3)
                .peek(e -> System.out.println("Filtered
value: " + e))
                .map(String::toUpperCase)
                .peek(e -> System.out.println("Mapped
value: " + e))
                .collect(Collectors.toList());
```

上面的例子输出：

```
Filtered value: three
Mapped value: THREE
Filtered value: four
Mapped value: FOUR
```

上面的例子我们输出了stream的中间值，方便我们的调试。

为什么只作为debug使用呢？我们再看一个例子：

```
Stream.of("one", "two", "three","four").peek(u ->
u.toUpperCase())
                .forEach(System.out::println);
```

上面的例子我们使用peek将element转换成为upper case。然后输出：

```
one
two
three
four
```

可以看到stream中的元素并没有被转换成大写格式。

再看一个map的对比：

```
Stream.of("one", "two", "three","four").map(u ->
u.toUpperCase())
                .forEach(System.out::println);
```

输出：

```
ONE
TWO
THREE
FOUR
```

可以看到map是真正的对元素进行了转换。

当然peek也有例外，假如我们Stream里面是一个对象会怎么样？

```
    @Data
    @AllArgsConstructor
    static class User{
        private String name;
    }
```

```
        List<User> userList=Stream.of(new User("a"),new
User("b"),new User("c")).peek(u-
>u.setName("kkk")).collect(Collectors.toList());
        log.info("{}",userList);
```

输出结果：

```
10:25:59.784 [main] INFO com.flydean.PeekUsage -
[PeekUsage.User(name=kkk), PeekUsage.User(name=kkk),
PeekUsage.User(name=kkk)]
```

我们看到如果是对象的话，实际的结果会被改变。

为什么peek和map有这样的区别呢？

我们看下peek和map的定义：

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

peek接收一个Consumer，而map接收一个Function。

Consumer是没有返回值的，它只是对Stream中的元素进行某些操作，但是操作之后的数据并不返回到Stream中，所以Stream中的元素还是原来的元素。

而Function是有返回值的，这意味着对于Stream的元素的所有操作都会作为新的结果返回到Stream中。

这就是为什么peek String不会发生变化而peek Object会发送变化的原因。

7. lambda表达式中的异常处理

java 8中引入了lambda表达式，lambda表达式可以让我们的代码更加简介，业务逻辑更加清晰，但是在lambda表达式中使用的Functional Interface并没有很好的处理异常，因为JDK提供的这些Functional Interface通常都是没有抛出异常的，这意味着需要我们自己手动来处理异常。

因为异常分为Unchecked Exception和checked Exception,我们分别来讨论。

7.1 处理Unchecked Exception

Unchecked exception也叫做RuntimeException，出现RuntimeException通常是因为我们的代码有问题。RuntimeException是不需要被捕获的。也就是说如果有RuntimeException，没有捕获也可以通过编译。

我们看一个例子：

```
List<Integer> integers = Arrays.asList(1,2,3,4,5);
        integers.forEach(i -> System.out.println(1 /
i));
```

这个例子是可以编译成功的，但是上面有一个问题，如果list中有一个0的话，就会抛出ArithmeticException。

虽然这个是一个Unchecked Exception，但是我们还是想处理一下：

```
        integers.forEach(i -> {
            try {
                System.out.println(1 / i);
            } catch (ArithmeticException e) {
                System.err.println(
                        "Arithmetic Exception occured :
" + e.getMessage());
            }
        });
```

上面的例子我们使用了try，catch来处理异常，简单但是破坏了lambda表达式的最佳实践。代码变得臃肿。

我们将try，catch移到一个wrapper方法中：

```
    static Consumer<Integer>
lambdaWrapper(Consumer<Integer> consumer) {
        return i -> {
            try {
                consumer.accept(i);
            } catch (ArithmeticException e) {
                System.err.println(
                        "Arithmetic Exception occured :
" + e.getMessage());
            }
        };
    }
```

则原来的调用变成这样：

```
integers.forEach(lambdaWrapper(i -> System.out.println(1
/ i)));
```

但是上面的wrapper固定了捕获ArithmeticException，我们再将其改编成一个更通用的类：

```
    static <T, E extends Exception> Consumer<T>
    consumerWrapperWithExceptionClass(Consumer<T>
consumer, Class<E> clazz) {

        return i -> {
            try {
                consumer.accept(i);
            } catch (Exception ex) {
                try {
                    E exCast = clazz.cast(ex);
                    System.err.println(
                            "Exception occured : " +
exCast.getMessage());
                } catch (ClassCastException ccEx) {
                    throw ex;
                }
            }
        };
    }
```

上面的类传入一个class，并将其cast到异常，如果能cast，则处理，否则抛出异常。

这样处理之后，我们这样调用：

```
integers.forEach(
                consumerWrapperWithExceptionClass(
                        i -> System.out.println(1 / i),
                        ArithmeticException.class));
```

7.2 处理checked Exception

checked Exception是必须要处理的异常，我们还是看个例子：

```
    static void throwIOException(Integer integer) throws
IOException {
    }
```

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
        integers.forEach(i -> throwIOException(i));
```

上面我们定义了一个方法抛出IOException，这是一个checked Exception，需要被处理，所以在下面的forEach中，程序会编译失败，因为没有处理相应的异常。

最简单的办法就是try，catch住，如下所示：

```
        integers.forEach(i -> {
            try {
                throwIOException(i);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        });
```

当然，这样的做法的坏处我们在上面已经讲过了，同样的，我们可以定义一个新的wrapper方法：

```
    static <T> Consumer<T> consumerWrapper(
            ThrowingConsumer<T, Exception>
throwingConsumer) {

        return i -> {
            try {
                throwingConsumer.accept(i);
            } catch (Exception ex) {
                throw new RuntimeException(ex);
            }
        };
    }
```


我们这样调用：

```
integers.forEach(consumerWrapper(i ->
throwIOException(i)));
```

我们也可以封装一下异常：

```
static <T, E extends Exception> Consumer<T>
consumerWrapperWithExceptionClass(
            ThrowingConsumer<T, E> throwingConsumer,
Class<E> exceptionClass) {

        return i -> {
            try {
                throwingConsumer.accept(i);
            } catch (Exception ex) {
                try {
                    E exCast = exceptionClass.cast(ex);
                    System.err.println(
```

```
                                "Exception occured : " +
exCast.getMessage());
                } catch (ClassCastException ccEx) {
                    throw new RuntimeException(ex);
                }
            }
        };
    }
```

然后这样调用：

```
integers.forEach(consumerWrapperWithExceptionClass(
                i -> throwIOException(i),
IOException.class));
```

8. stream中throw Exception

之前的文章我们讲到，在stream中处理异常，需要将checked exception转换为unchecked exception来处理。

我们是这样做的：

```
    static <T> Consumer<T> consumerWrapper(
            ThrowingConsumer<T, Exception>
throwingConsumer) {

        return i -> {
            try {
                throwingConsumer.accept(i);
            } catch (Exception ex) {
                throw new RuntimeException(ex);
            }
        };
    }
```

将异常捕获，然后封装成为RuntimeException。

封装成RuntimeException感觉总是有那么一点点问题，那么有没有什么更好的办法？

8.1 throw小诀窍

java的类型推断大家应该都知道，如果是<T extends Throwable> 这样的形式，那么T将会被认为是RuntimeException！

我们看下例子：

```
public class RethrowException {

    public static <T extends Exception, R> R
throwException(Exception t) throws T {
        throw (T) t; // just throw it, convert checked
exception to unchecked exception
    }

}
```

上面的类中，我们定义了一个throwException方法，接收一个Exception参数，将其转换为T，这里的T就是unchecked exception。

接下来看下具体的使用：

```
@Slf4j
public class RethrowUsage {

    public static void main(String[] args) {
        try {
            throwIOException();
        } catch (IOException e) {
           log.error(e.getMessage(),e);
            RethrowException.throwException(e);
        }
    }

    static void throwIOException() throws IOException{
        throw new IOException("io exception");
    }
}
```


上面的例子中，我们将一个IOException转换成了一个unchecked exception。

9. stream中Collectors的用法

在java stream中，我们通常需要将处理后的stream转换成集合类，这个时候就需要用到stream.collect方法。collect方法需要传入一个Collector类型，要实现Collector还是很麻烦的，需要实现好几个接口。

于是java提供了更简单的Collectors工具类来方便我们构建Collector。

下面我们将会具体讲解Collectors的用法。

假如我们有这样两个list：

```
List<String> list = Arrays.asList("jack", "bob",
"alice", "mark");
List<String> duplicateList = Arrays.asList("jack",
"jack", "alice", "mark");
```

上面一个是无重复的list，一个是带重复数据的list。接下来的例子我们会用上面的两个list来讲解Collectors的用法。

9.1 Collectors.toList()

```
List<String> listResult =
list.stream().collect(Collectors.toList());
        log.info("{}",listResult);
```

将stream转换为list。这里转换的list是ArrayList，如果想要转换成特定的list，需要使用toCollection方法。

9.2 Collectors.toSet()

```
Set<String> setResult =
list.stream().collect(Collectors.toSet());
        log.info("{}",setResult);
```

toSet将Stream转换成为set。这里转换的是HashSet。如果需要特别指定set，那么需要使用toCollection方法。

因为set中是没有重复的元素，如果我们使用duplicateList来转换的话，会发现最终结果中只有一个jack。

```
Set<String> duplicateSetResult =
duplicateList.stream().collect(Collectors.toSet());
        log.info("{}",duplicateSetResult);
```

9.3 Collectors.toCollection()

上面的toMap,toSet转换出来的都是特定的类型，如果我们需要自定义，则可以使用toCollection()

```
List<String> custListResult =
list.stream().collect(Collectors.toCollection(LinkedList
::new));
        log.info("{}",custListResult);
```

上面的例子，我们转换成了LinkedList。

9.4 Collectors.toMap()

toMap接收两个参数，第一个参数是keyMapper，第二个参数是valueMapper:

```
Map<String, Integer> mapResult = list.stream()

.collect(Collectors.toMap(Function.identity(),
String::length));
        log.info("{}",mapResult);
```

如果stream中有重复的值，则转换会报IllegalStateException异常:

```
Map<String, Integer> duplicateMapResult =
duplicateList.stream()

.collect(Collectors.toMap(Function.identity(),
String::length));
```

怎么解决这个问题呢？我们可以这样:

```
Map<String, Integer> duplicateMapResult2 =
duplicateList.stream()

.collect(Collectors.toMap(Function.identity(),
String::length, (item, identicalItem) -> item));
        log.info("{}",duplicateMapResult2);
```

在toMap中添加第三个参数mergeFunction，来解决冲突的问题。

9.5 Collectors.collectingAndThen()

collectingAndThen允许我们对生成的集合再做一次操作。

```
List<String> collectAndThenResult = list.stream()

.collect(Collectors.collectingAndThen(Collectors.toList(
), l -> {return new ArrayList<>(l);}));
        log.info("{}",collectAndThenResult);
```

9.6 Collectors.joining()

Joining用来连接stream中的元素：

```
String joinResult =
list.stream().collect(Collectors.joining());
        log.info("{}",joinResult);
        String joinResult1 =
list.stream().collect(Collectors.joining(" "));
        log.info("{}",joinResult1);
        String joinResult2 =
list.stream().collect(Collectors.joining(" ",
"prefix","suffix"));
        log.info("{}",joinResult2);
```

可以不带参数，也可以带一个参数，也可以带三个参数，根据我们的需要进行选择。

9.7 Collectors.counting()

counting主要用来统计stream中元素的个数：

```
Long countResult =
list.stream().collect(Collectors.counting());
        log.info("{}",countResult);
```

9.8 Collectors.summarizingDouble/Long/Int()

SummarizingDouble/Long/Int为stream中的元素生成了统计信息，返回的结果是一个统计类：

```
IntSummaryStatistics intResult = list.stream()

.collect(Collectors.summarizingInt(String::length));
        log.info("{}",intResult);
```

输出结果：

```
22:22:35.238 [main] INFO com.flydean.CollectorUsage -
IntSummaryStatistics{count=4, sum=16, min=3,
average=4.000000, max=5}
```

9.9 Collectors.averagingDouble/Long/Int()

averagingDouble/Long/Int()对stream中的元素做平均：

```
Double averageResult =
list.stream().collect(Collectors.averagingInt(String::le
ngth));
        log.info("{}",averageResult);
```

9.10 Collectors.summingDouble/Long/Int()

summingDouble/Long/Int()对stream中的元素做sum操作：

```
Double summingResult =
list.stream().collect(Collectors.summingDouble(String::l
ength));
        log.info("{}",summingResult);
```

9.11 Collectors.maxBy()/minBy()

maxBy()/minBy()根据提供的Comparator，返回stream中的最大或者最小值：

```
Optional<String> maxByResult =
list.stream().collect(Collectors.maxBy(Comparator.natura
lOrder()));
        log.info("{}",maxByResult);
```

9.12 Collectors.groupingBy()

GroupingBy根据某些属性进行分组，并返回一个Map：

```
Map<Integer, Set<String>> groupByResult = list.stream()

.collect(Collectors.groupingBy(String::length,
Collectors.toSet()));
        log.info("{}",groupByResult);
```

9.13 Collectors.partitioningBy()

PartitioningBy是一个特别的groupingBy，PartitioningBy返回一个Map，这个Map是以boolean值为key，从而将stream分成两部分，一部分是匹配PartitioningBy条件的，一部分是不满足条件的：

```
 Map<Boolean, List<String>> partitionResult =
list.stream()
                .collect(Collectors.partitioningBy(s ->
s.length() > 3));
        log.info("{}",partitionResult);
```

看下运行结果：

```
22:39:37.082 [main] INFO com.flydean.CollectorUsage -
{false=[bob], true=[jack, alice, mark]}
```

结果被分成了两部分。

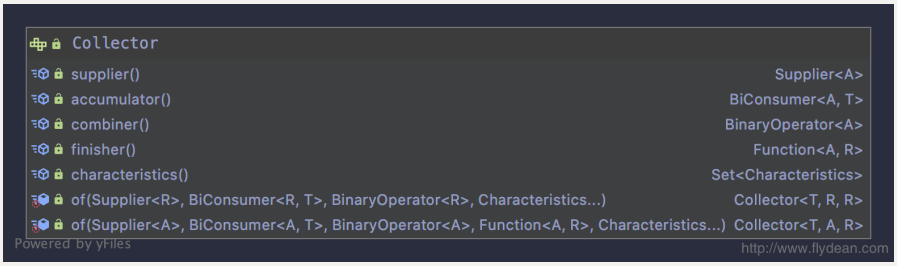
10. 创建一个自定义的collector

在之前的java collectors文章里面，我们讲到了stream的collect方法可以调用Collectors里面的toList()或者toMap()方法，将结果转换为特定的集合类。

今天我们介绍一下怎么自定义一个Collector。

10.1 Collector介绍

我们先看一下Collector的定义：



Collector接口需要实现supplier(),accumulator(),combiner(),finisher(),characteristics()这5个接口。

同时Collector也提供了两个静态of方法来方便我们创建一个Collector实例。

我们可以看到两个方法的参数跟Collector接口需要实现的接口是一一对应的。

下面分别解释一下这几个参数：

- supplier

Supplier是一个函数，用来创建一个新的可变的集合。换句话说Supplier用来创建一个初始的集合。

- accumulator

accumulator定义了累加器，用来将原始元素添加到集合中。

- combiner

combiner用来将两个集合合并成一个。

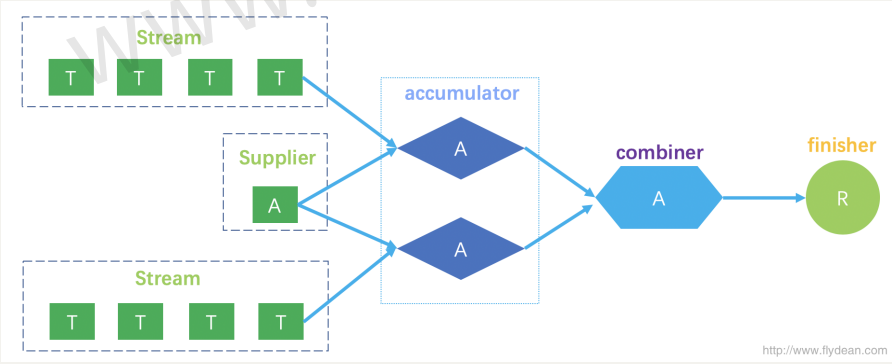
- finisher

finisher将集合转换为最终的集合类型。

- characteristics

characteristics表示该集合的特征。这个不是必须的参数。

Collector定义了三个参数类型，T是输入元素的类型，A是reduction operation的累加类型也就是Supplier的初始类型，R是最终的返回类型。 我们画个图来看一下这些类型之间的转换关系：



有了这几个参数，我们接下来看看怎么使用这些参数来构造一个自定义Collector。

10.2 自定义Collector

我们利用Collector的of方法来创建一个不变的Set：

```
    public static <T> Collector<T, Set<T>, Set<T>>
toImmutableSet() {
        return Collector.of(HashSet::new, Set::add,
                (left, right) -> {
                    left.addAll(right);
                    return left;
                }, Collections::unmodifiableSet);
    }
```

上面的例子中，我们HashSet::new作为supplier，Set::add作为accumulator，自定义了一个方法作为combiner，最后使用Collections::unmodifiableSet将集合转换成不可变集合。

上面我们固定使用HashSet::new作为初始集合的生成方法，实际上，上面的方法可以更加通用：

```
    public static <T, A extends Set<T>> Collector<T, A,
Set<T>> toImmutableSet(
            Supplier<A> supplier) {

        return Collector.of(
                supplier,
                Set::add, (left, right) -> {
                    left.addAll(right);
                    return left;
                }, Collections::unmodifiableSet);
    }
```


上面的方法，我们将supplier提出来作为一个参数，由外部来定义。

看下上面两个方法的测试：

```
    @Test
    public void toImmutableSetUsage(){
        Set<String>
stringSet1=Stream.of("a","b","c","d")

.collect(ImmutableSetCollector.toImmutableSet());
        log.info("{}",stringSet1);

        Set<String>
stringSet2=Stream.of("a","b","c","d")

.collect(ImmutableSetCollector.toImmutableSet(LinkedHash
Set::new));
        log.info("{}",stringSet2);
    }
```

输出：

```
INFO com.flydean.ImmutableSetCollector - [a, b, c, d]
INFO com.flydean.ImmutableSetCollector - [a, b, c, d]
```

11. stream reduce详解和误区

Stream API提供了一些预定义的reduce操作，比如count(), max(), min(), sum()等。如果我们需要自己写reduce的逻辑，则可以使用reduce方法。

本文将会详细分析一下reduce方法的使用，并给出具体的例子。

11.1 reduce详解

Stream类中有三种reduce，分别接受1个参数，2个参数，和3个参数，首先来看一个参数的情况：

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

该方法接受一个BinaryOperator参数，BinaryOperator是一个@FunctionalInterface,需要实现方法：

```
R apply(T t, U u);
```

accumulator告诉reduce方法怎么去累计stream中的数据。

举个例子：

```
List<Integer> intList = Arrays.asList(1,2,3);
        Optional<Integer>
result1=intList.stream().reduce(Integer::sum);
        log.info("{}",result1);
```

上面的例子输出结果：

```
com.flydean.ReduceUsage - Optional[6]
```

一个参数的例子很简单。这里不再多说。

接下来我们再看一下两个参数的例子：

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

这个方法接收两个参数：identity和accumulator。多出了一个参数identity。

也许在有些文章里面有人告诉你identity是reduce的初始化值，可以随便指定，如下所示：

```
Integer result2=intList.stream().reduce(100,
Integer::sum);
        log.info("{}",result2);
```


上面的例子，我们计算的值是106。

如果我们将stream改成parallelStream：

```
Integer result3=intList.parallelStream().reduce(100,
Integer::sum);
        log.info("{}",result3);
```

得出的结果就是306。

为什么是306呢？因为在并行计算的时候，每个线程的初始累加值都是100，最后3个线程加出来的结果就是306。

并行计算和非并行计算的结果居然不一样，这肯定不是JDK的问题，我们再看一下JDK中对identity的说明：

> *identity必须是accumulator函数的一个identity，也就是说必须满足：对于所有的t,都必须满足 accumulator.apply(identity, t) == t*

所以这里我们传入100是不对的，因为sum（100+1）！= 1。

这里sum方法的identity只能是0。

如果我们用0作为identity,则stream和parallelStream计算出的结果是一样的。这就是identity的真正意图。

下面再看一下三个参数的方法：

```
<U> U reduce(U identity,
                  BiFunction<U, ? super T, U>
accumulator,
                  BinaryOperator<U> combiner);
```

和前面的方法不同的是，多了一个combiner，这个combiner用来合并多线程计算的结果。

> *同样的，identity需要满足combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)*

大家可能注意到了为什么accumulator的类型是BiFunction而combiner的类型是BinaryOperator?

```
public interface BinaryOperator<T> extends
BiFunction<T,T,T>
```

BinaryOperator是BiFunction的子接口。BiFunction中定义了要实现的apply方法。

其实reduce底层方法的实现只用到了apply方法，并没有用到接口中其他的方法，所以我猜测这里的不同只是为了简单的区分。

虽然reduce是一个很常用的方法，但是大家一定要遵循identity的规范，并不是所有的identity都是合适的。

12. stream中的Spliterator

Spliterator是在java 8引入的一个接口，它通常和stream一起使用，用来遍历和分割序列。

只要用到stream的地方都需要Spliterator，比如List，Collection，IO channel等等。

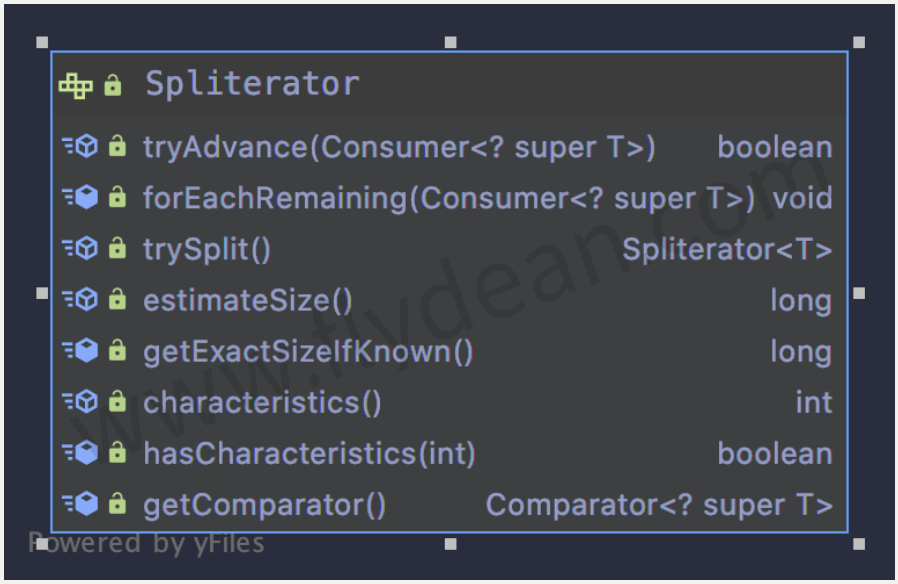
我们先看一下Collection中stream方法的定义：

```
default Stream<E> stream() {
        return StreamSupport.stream(spliterator(),
false);
    }
```

```
default Stream<E> parallelStream() {
        return StreamSupport.stream(spliterator(),
true);
    }
```

我们可以看到，不管是并行stream还是非并行stream，都是通过StreamSupport来构造的，并且都需要传入一个spliterator的参数。

好了，我们知道了spliterator是做什么的之后，看一下它的具体结构：



spliterator有四个必须实现的方法，我们接下来进行详细的讲解。

12.1 tryAdvance

tryAdvance就是对stream中的元素进行处理的方法，如果元素存在，则对他进行处理，并返回true，否则返回false。

如果我们不想处理stream后续的元素，则在tryAdvance中返回false即可，利用这个特征，我们可以中断stream的处理。这个例子我将会在后面的文章中讲到。

12.2 trySplit

trySplit尝试对现有的stream进行分拆，一般用在parallelStream的情况，因为在并发stream下，我们需要用多线程去处理stream的不同元素，trySplit就是对stream中元素进行分拆处理的方法。

理想情况下trySplit应该将stream拆分成数目相同的两部分才能最大提升性能。

12.3 estimateSize

estimateSize表示Spliterator中待处理的元素，在trySplit之前和之后一般是不同的，后面我们会在具体的例子中说明。

12.4 characteristics

characteristics表示这个Spliterator的特征，Spliterator有8大特征：

```
public static final int ORDERED    = 0x00000010;//表示元素是有序的（每一次遍历结果相同）
public static final int DISTINCT   = 0x00000001;//表示元素不重复
public static final int SORTED     = 0x00000004;//表示元素是按一定规律进行排列（有指定比较器）
public static final int SIZED      = 0x00000040;//表示大小是固定的
public static final int NONNULL    = 0x00000100;//表示没有null元素
public static final int IMMUTABLE  = 0x00000400;//表示元素不可变
public static final int CONCURRENT = 0x00001000;//表示迭代器可以多线程操作
public static final int SUBSIZED   = 0x00004000;//表示子Spliterators都具有SIZED特性
```

一个Spliterator可以有多个特征，多个特征进行or运算，最后得到最终的characteristics。

12.5 举个例子

上面我们讨论了Spliterator一些关键方法，现在我们举一个具体的例子：

```
@AllArgsConstructor
@Data
public class CustBook {
    private String name;

}
```

先定义一个CustBook类，里面放一个name变量。

定义一个方法，来生成一个CustBook的list：

```
    public static List<CustBook> generateElements() {
        return Stream.generate(() -> new CustBook("cust
book"))
                .limit(1000)
                .collect(Collectors.toList());
    }
```

我们定义一个call方法，在call方法中调用了tryAdvance方法，传入了我们自定义的处理方法。这里我们修改book的name,并附加额外的信息。

```
    public String call(Spliterator<CustBook>
spliterator) {
        int current = 0;
        while (spliterator.tryAdvance(a ->
a.setName("test name"
                .concat("- add new name")))) {
            current++;
        }

        return Thread.currentThread().getName() + ":" +
current;
    }
```


最后，写一下测试方法：

```
    @Test
    public void useTrySplit(){
        Spliterator<CustBook> split1 =
SpliteratorUsage.generateElements().spliterator();
        Spliterator<CustBook> split2 =
split1.trySplit();

        log.info("before tryAdvance:
{}",split1.estimateSize());
        log.info("Characteristics
{}",split1.characteristics());
        log.info(call(split1));
        log.info(call(split2));
        log.info("after tryAdvance
{}",split1.estimateSize());
    }
```

运行的结果如下：

```
23:10:08.852 [main] INFO com.flydean.SpliteratorUsage -
before tryAdvance: 500
23:10:08.857 [main] INFO com.flydean.SpliteratorUsage -
Characteristics 16464
23:10:08.858 [main] INFO com.flydean.SpliteratorUsage -
main:500
23:10:08.858 [main] INFO com.flydean.SpliteratorUsage -
main:500
23:10:08.858 [main] INFO com.flydean.SpliteratorUsage -
after tryAdvance 0
```

List总共有1000条数据，调用一次trySplit之后，将List分成了两部分，每部分500条数据。

注意，在tryAdvance调用之后，estimateSize变为0，表示所有的元素都已经被处理完毕。

再看一下这个Characteristics=16464，转换为16进制：Ox4050 = ORDERED or SIZED or SUBSIZED 这三个的或运算。

这也是ArrayList的基本特征。

13. break stream的foreach

我们通常需要在java stream中遍历处理里面的数据，其中foreach是最最常用的方法。

但是有时候我们并不想处理完所有的数据，或者有时候Stream可能非常的长，或者根本就是无限的。

一种方法是先filter出我们需要处理的数据，然后再foreach遍历。

那么我们如何直接break这个stream呢？今天本文重点讲解一下这个问题。

13.1 使用Spliterator

上篇文章我们在讲Spliterator的时候提到了，在tryAdvance方法中，如果返回false，则Spliterator将会停止处理后续的元素。

通过这个思路，我们可以创建自定义Spliterator。

假如我们有这样一个stream：

```
Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5, 6, 7, 8,
9, 10);
```

我们想定义一个操作，当x > 5的时候就停止。

我们定义一个通用的Spliterator：

```
public class CustomSpliterator<T> extends
Spliterators.AbstractSpliterator<T>  {

    private Spliterator<T> splitr;
    private Predicate<T> predicate;
    private volatile boolean isMatched = true;

    public CustomSpliterator(Spliterator<T> splitr,
Predicate<T> predicate) {
        super(splitr.estimateSize(), 0);
        this.splitr = splitr;
        this.predicate = predicate;
    }

    @Override
    public synchronized boolean tryAdvance(Consumer<?
super T> consumer) {
        boolean hadNext = splitr.tryAdvance(elem -> {
            if (predicate.test(elem) && isMatched) {
                consumer.accept(elem);
            } else {
                isMatched = false;
            }
        });
        return hadNext && isMatched;
    }
}
```

在上面的类中，predicate是我们将要传入的判断条件，我们重写了tryAdvance，通过将predicate.test(elem)加入判断条件，从而当条件不满足的时候返回false.

看下怎么使用：

```
@Slf4j
public class CustomSpliteratorUsage {

    public static <T> Stream<T> takeWhile(Stream<T>
stream, Predicate<T> predicate) {
```

```
        CustomSpliterator<T> customSpliterator = new
CustomSpliterator<>(stream.spliterator(), predicate);
        return StreamSupport.stream(customSpliterator,
false);
    }

    public static void main(String[] args) {
        Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5,
6, 7, 8, 9, 10);
        List<Integer> result =
          takeWhile(ints, x -> x < 5 )
                        .collect(Collectors.toList());
        log.info(result.toString());
    }
}
```

我们定义了一个takeWhile方法，接收Stream和predicate条件。

只有当predicate条件满足的时候才会继续，我们看下输出的结果：

```
[main] INFO com.flydean.CustomSpliteratorUsage - [1, 2,
3, 4]
```

13.2 自定义forEach方法

除了使用Spliterator，我们还可以自定义forEach方法来使用自己的遍历逻辑：

```
public class CustomForEach {

    public static class Breaker {
        private volatile boolean shouldBreak = false;

        public void stop() {
            shouldBreak = true;
        }

        boolean get() {
            return shouldBreak;
        }
    }

    public static <T> void forEach(Stream<T> stream,
BiConsumer<T, Breaker> consumer) {
        Spliterator<T> spliterator =
stream.spliterator();
        boolean hadNext = true;
```

```
        Breaker breaker = new Breaker();

        while (hadNext && !breaker.get()) {
            hadNext = spliterator.tryAdvance(elem -> {
                consumer.accept(elem, breaker);
            });
        }
    }
}
```

上面的例子中，我们在forEach中引入了一个外部变量，通过判断这个外部变量来决定是否进入spliterator.tryAdvance方法。

看下怎么使用：

```
@Slf4j
public class CustomForEachUsage {

    public static void main(String[] args) {
        Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5,
6, 7, 8, 9, 10);
        List<Integer> result = new ArrayList<>();
        CustomForEach.forEach(ints, (elem, breaker) -> {
            if (elem >= 5 ) {
                breaker.stop();
            } else {
                result.add(elem);
            }
        });
        log.info(result.toString());
    }
}
```

上面我们用新的forEach方法，并通过判断条件来重置判断flag，从而达到break stream的目的。

14. predicate chain的使用

Predicate是一个FunctionalInterface，代表的方法需要输入一个参数，返回boolean类型。通常用在stream的filter中，表示是否满足过滤条件。

```
    boolean test(T t);
```

14.1 基本使用

我们先看下在stream的filter中怎么使用Predicate：

```
    @Test
    public void basicUsage(){
        List<String>
stringList=Stream.of("a","b","c","d").filter(s ->
s.startsWith("a")).collect(Collectors.toList());
        log.info("{}",stringList);
    }
```

上面的例子很基础了，这里就不多讲了。

14.2 使用多个Filter

如果我们有多个Predicate条件，则可以使用多个filter来进行过滤：

```
    public void multipleFilters(){
        List<String>
stringList=Stream.of("a","ab","aac","ad").filter(s ->
s.startsWith("a"))
                .filter(s -> s.length()>1)
                .collect(Collectors.toList());
        log.info("{}",stringList);
    }
```


上面的例子中，我们又添加了一个filter，在filter又添加了一个Predicate。

14.3 使用复合Predicate

Predicate的定义是输入一个参数，返回boolean值，那么如果有多个测试条件，我们可以将其合并成一个test方法：

```
    @Test
    public void complexPredicate(){
        List<String>
stringList=Stream.of("a","ab","aac","ad")
                .filter(s -> s.startsWith("a") &&
 s.length()>1)
                .collect(Collectors.toList());
        log.info("{}",stringList);
    }
```

上面的例子中，我们把s.startsWith("a") && s.length()>1 作为test的实现。

14.4 组合Predicate

Predicate虽然是一个interface，但是它有几个默认的方法可以用来实现Predicate之间的组合操作。

比如：Predicate.and(), Predicate.or(), 和 Predicate.negate()。

下面看下他们的例子：

```
@Test
    public void combiningPredicate(){
        Predicate<String> predicate1 = s ->
s.startsWith("a");
        Predicate<String> predicate2 =  s -> s.length()
> 1;
        List<String> stringList1 =
Stream.of("a","ab","aac","ad")
                .filter(predicate1.and(predicate2))
                .collect(Collectors.toList());
        log.info("{}",stringList1);

        List<String> stringList2 =
Stream.of("a","ab","aac","ad")
                .filter(predicate1.or(predicate2))
                .collect(Collectors.toList());
        log.info("{}",stringList2);

        List<String> stringList3 =
Stream.of("a","ab","aac","ad")

.filter(predicate1.or(predicate2.negate()))
                .collect(Collectors.toList());
        log.info("{}",stringList3);

    }
```

实际上，我们并不需要显示的assign一个predicate，只要是满足predicate接口的lambda表达式都可以看做是一个predicate。同样可以调用and，or和negate操作：

```
List<String> stringList4 =
Stream.of("a","ab","aac","ad")
                .filter(((Predicate<String>)a ->
a.startsWith("a"))
                        .and(a -> a.length() > 1))
                .collect(Collectors.toList());
        log.info("{}",stringList4);
```

14.5 Predicate的集合操作

如果我们有一个Predicate集合，我们可以使用reduce方法来对其进行合并运算：

```
@Test
    public void combiningPredicateCollection(){
        List<Predicate<String>> allPredicates = new
ArrayList<>();
        allPredicates.add(a -> a.startsWith("a"));
        allPredicates.add(a -> a.length() > 1);

        List<String> stringList =
Stream.of("a","ab","aac","ad")
                .filter(allPredicates.stream().reduce(x-
>true, Predicate::and))
                .collect(Collectors.toList());
        log.info("{}",stringList);
    }
```


上面的例子中，我们调用reduce方法，对集合中的Predicate进行了and操作。

15. 中构建无限的stream

在java中，我们可以将特定的集合转换成为stream，那么在有些情况下，比如测试环境中，我们需要构造一定数量元素的stream，需要怎么处理呢？

这里我们可以构建一个无限的stream，然后调用limit方法来限定返回的数目。

15.1 基本使用

先看一个使用Stream.iterate来创建无限Stream的例子：

```
    @Test
    public void infiniteStream(){
        Stream<Integer> infiniteStream =
Stream.iterate(0, i -> i + 1);
        List<Integer> collect = infiniteStream
                .limit(10)
                .collect(Collectors.toList());
        log.info("{}",collect);
    }
```

上面的例子中，我们通过调用Stream.iterate方法，创建了一个0，1，2，3，4....的无限stream。

然后调用limit(10)来获取其中的前10个。最后调用collect方法将其转换成为一个集合。

看下输出结果：

```
INFO com.flydean.InfiniteStreamUsage - [0, 1, 2, 3, 4,
5, 6, 7, 8, 9]
```

15.2 自定义类型

如果我们想输出自定义类型的集合，该怎么处理呢？

首先，我们定义一个自定义类型：

```
@Data
@AllArgsConstructor
public class IntegerWrapper {
    private Integer integer;
}
```

然后利用Stream.generate的生成器来创建这个自定义类型：

```
    public static IntegerWrapper generateCustType(){
        return new IntegerWrapper(new
Random().nextInt(100));
    }

    @Test
    public void infiniteCustType(){
        Supplier<IntegerWrapper> randomCustTypeSupplier
= InfiniteStreamUsage::generateCustType;
```

```
        Stream<IntegerWrapper> infiniteStreamOfCustType
= Stream.generate(randomCustTypeSupplier);

        List<IntegerWrapper> collect =
infiniteStreamOfCustType
                .skip(10)
                .limit(10)
                .collect(Collectors.toList());
        log.info("{}",collect);
    }
```

看下输出结果：

```
INFO com.flydean.InfiniteStreamUsage -
[IntegerWrapper(integer=46), IntegerWrapper(integer=42),
IntegerWrapper(integer=67), IntegerWrapper(integer=11),
IntegerWrapper(integer=14), IntegerWrapper(integer=80),
IntegerWrapper(integer=15), IntegerWrapper(integer=19),
IntegerWrapper(integer=72), IntegerWrapper(integer=41)]
```

16. 自定义parallelStream的thread pool

之前我们讲到parallelStream的底层使用到了ForkJoinPool来提交任务的，默认情况下ForkJoinPool为每一个处理器创建一个线程，parallelStream如果没有特别指明的情况下，都会使用这个共享线程池来提交任务。

那么在特定的情况下，我们想使用自定义的ForkJoinPool该怎么处理呢？

16.1 通常操作

假如我们想做一个从1到1000的加法，我们可以用并行stream这样做：

```
List<Integer> integerList=
IntStream.range(1,1000).boxed().collect(Collectors.toLis
t());
        ForkJoinPool customThreadPool = new
ForkJoinPool(4);

        Integer total=
integerList.parallelStream().reduce(0, Integer::sum);
        log.info("{}",total);
```

输出结果：

```
INFO com.flydean.CustThreadPool - 499500
```

16.2 使用自定义ForkJoinPool

上面的例子使用的共享的thread pool。我们看下怎么使用自定义的thread pool来提交并行stream：

```
List<Integer> integerList=
IntStream.range(1,1000).boxed().collect(Collectors.toLis
t());

ForkJoinPool customThreadPool = new ForkJoinPool(4);
        Integer actualTotal = customThreadPool.submit(
                () ->
integerList.parallelStream().reduce(0,
Integer::sum)).get();
        log.info("{}",actualTotal);
```

上面的例子中，我们定义了一个4个线程的ForkJoinPool，并使用它来提交了这个parallelStream。

输出结果：

```
INFO com.flydean.CustThreadPool - 499500
```

如果不想使用公共的线程池，则可以使用自定义的ForkJoinPool来提交。

17. 总结

本文统一介绍了Stream和lambda表达式的使用，涵盖了Stream和lambda表达式的各个小的细节，希望大家能够喜欢。

本文的代码https://github.com/ddean2009/learn-java-streams/

> *最通俗的解读，最深刻的干货，最简洁的教程，众多你不知道的小技巧等你来发现！*
>
> *欢迎关注我的公众号:「程序那些事」,懂技术，更懂你！*