**【9.13】JDK8新特性**

**Lambda表达式**

用来替代匿名函数，可以将一个函数赋值给一个变量作为参数传入另一个函数，java的闭包

原则：可推导就是可省略，比如说参数类型，返回值

// 1. 不需要参数,返回值为 5 {}只有一行代码，可以省略

() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值，()只有一个参数可以省略

x -> 2 \* x

// 3. 接受2个参数(数字),并返回他们的差值

(x, y) -> x – y

// 4. 接收2个int型整数,返回他们的和

(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)

(String s) -> System.out.print(s)

**语法**

Interface var = (x,y) -> {}

该接口只能有一个需要被实现的方法，小括号中参数取决于Interface 的接口方法的参数，没有参数则为空，{}中为方法的实现内容，如果内容只有一行代码，{}可以省略。实际上就是匿名函数

Runnable run = new Runnable(){

@Override

publicvoidrun(){

System.out.println("常规写法");

}

};

Runnable run1 = () -> {System.out.println("lambda");};//{}中只有一条语句时，{}可以省略

//匿名函数的访问权限可以省略（跟接收变量的作用域保持一致，返回值和参数类型都可以编译器自动判断。）

只有一个抽象方法需要被实现的接口，称为**“函数式接口”**，为了避免后续被人在该接口中添加方法，导致规则被破坏，可以在该接口上加一个声明@FunctionalInterface，这样该接口就无法添加新的接口函数了

**变量作用域**

lambda 表达式只能引用final 类型的外层局部变量，就是说不能在 lambda 内部修改定义在域外的局部变量，否则会编译错误。与匿名函数同理

lambda 表达式的局部变量可以不用声明为 final，但是必须不可被后面的代码修改（即隐性的具有 final 的语义）

int num = 1;

Converter<Integer, String> s = (param) -> System.out.println(String.valueOf(param + num));

s.convert(2);

num = 5;

//报错信息：Local variable num defined in an enclosing scope must be final or effectively final

//在 Lambda 表达式当中不允许声明一个与局部变量同名的参数或者局部变量。

String first = "";

Comparator<String> comparator = (first, second) -> Integer.compare(first.length(), second.length()); //编译会出错

**方法引用**

若Lambda体中的内容有方法已经实现了，我们可以使用“方法引用”，可以理解为方法引用是lambda表达式的另外一种表达形式

主要有三种语法格式：

* 对象 :: 实例方法名
* 类 :: 静态方法名
* 类 :: 实例方法名

被引用的方法的参数和返回值必须和要实现的抽象方法的参数和返回值一致

**静态方法引用**

//格式：Classname :: staticMethodName 和静态方法调用相比，只是把 . 换为 ::

String::valueOf 等价于lambda表达式 (s) -> String.valueOf(s)

Math::pow 等价于lambda表达式 (x, y) -> Math.pow(x, y);

**实例对象方法引用**

//格式：instanceReference::methodName

class ComparisonProvider{

public int compareByName(Person a, Person b){

return a.getName().compareTo(b.getName());

}

public int compareByAge(Person a, Person b){

return a.getBirthday().compareTo(b.getBirthday());

}

}

ComparisonProvider myComparisonProvider = new ComparisonProvider();

Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);

**超类上的实例方法引用**

//格式：super::methodName

//还可以使用this

**泛型类和泛型方法引用**

public interface MyFunc<T> {

int func(T[] als, T v);

}

public class MyArrayOps {

public static <T> int countMatching(T[] vals, T v) {

int count = 0;

for (int i = 0; i < vals.length; i++) {

if (vals[i] == v) count++;

}

return count;

}

}

public class GenericMethodRefDemo {

public static <T> int myOp(MyFunc<T> f, T[] vals, T v) {

return f.func(vals, v);

}

public static void main(String[] args){

Integer[] vals = {1, 2, 3, 4, 2, 3, 4, 4, 5};

String[] strs = {"One", "Two", "Three", "Two"};

int count;

count=myOp(MyArrayOps::<Integer>countMatching, vals, 4);

System.out.println("vals contains "+count+" 4s");

count=myOp(MyArrayOps::<String>countMatching, strs, "Two");

System.out.println("strs contains "+count+" Twos");

}

}

**构造器引用**

通过函数式接口实例化类时可以用构造器引用，引用到的是方法参数个数和类型匹配的构造器

//格式：ClassName :: new，调用默认构造器。

//lambda方式

Supplier<Passenger> supplier1 = () -> new Passenger();

//构造器引用:通过类型推断，引用无参构造器

Supplier<Passenger> supplier2 = Passenger::new;

//lambda方式

BiFunction<String, String, Passenger> function1 = (x, y) -> new Passenger(x, y);

//构造器引用:通过类型推断，引用有两个String参数的构造器

BiFunction<String, String, Passenger> function2 = Passenger::new;

**数组引用**

//lambda方式

Function<Integer, String[]> fun1 = (x) -> new String[x];

String[] strs1 = fun1.apply(10);

//数组引用

Function<Integer, String[]> fun2 = String[]::new;

String[] strs2 = fun2.apply(10);

**Stream**

A sequence of elements supporting sequential and parallel aggregate operations.

1、Stream是元素的集合，这点让Stream看起来用些类似Iterator；

2、可以支持顺序和并行的对原Stream进行汇聚的操作；

特性：

* 不存储数据
* 不改变源数据
* 延迟执行

使用步骤：

* 创建Stream数据源；
* 数据处理，转换Stream，每次转换原有Stream对象不改变，返回一个新的Stream对象（**可以有多次转换**）；
* 对Stream进行聚合（Reduce）操作，获取想要的结果；

**创建数据源：**

1、Collection.stream(); 从集合获取流。

2、Collection.parallelStream(); 从集合获取并行流。

3、Arrays.stream(T array) or Stream.of(); 从数组获取流。

4、BufferedReader.lines(); 从输入流中获取流。

5、IntStream.of() ; 从静态方法中获取流。

6、Stream.generate(); 自己生成流

@Test

public void createStream() throws FileNotFoundException {

List<String> nameList = Arrays.asList("Darcy", "Chris", "Linda", "Sid", "Kim", "Jack", "Poul", "Peter");

String[] nameArr = {"Darcy", "Chris", "Linda", "Sid", "Kim", "Jack", "Poul", "Peter"};

// 集合获取 Stream 流

Stream<String> nameListStream = nameList.stream();

// 集合获取并行 Stream 流

Stream<String> nameListStream2 = nameList.parallelStream();

// 数组获取 Stream 流

Stream<String> nameArrStream = Stream.of(nameArr);

// 数组获取 Stream 流

Stream<String> nameArrStream1 = Arrays.stream(nameArr);

// 文件流获取 Stream 流

BufferedReader bufferedReader = new BufferedReader(new FileReader("README.md"));

Stream<String> linesStream = bufferedReader.lines();

// 从静态方法获取流操作

IntStream rangeStream = IntStream.range(1, 10);

rangeStream.limit(10).forEach(num -> System.out.print(num+","));

System.out.println();

IntStream intStream = IntStream.of(1, 2, 3, 3, 4);

intStream.forEach(num -> System.out.print(num+","));

}

**数据处理/转换**

中间操作，可以有多个，返回的是一个新的stream对象，惰性计算，只有在开始收集结果时中间操作才会生效。

map (mapToInt, flatMap )：把对象映射成另一种对象

@Test

public void mapTest() {

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

// 映射成 2倍数字

List<Integer> collect = numberList.stream()

.map(number -> number \* 2)

.collect(Collectors.toList());

collect.forEach(number -> System.out.print(number + ","));

System.out.println();

numberList.stream()

.map(number -> "数字 " + number + ",")

.forEach(number -> System.out.println(number));

}

@Test

public void flatMapTest() {

Stream<List<Integer>> inputStream = Stream.of(

Arrays.asList(1),

Arrays.asList(2, 3),

Arrays.asList(4, 5, 6)

);

List<Integer> collect = inputStream

.flatMap((childList) -> childList.stream())

.collect(Collectors.toList());

collect.forEach(number -> System.out.print(number + ","));

}

// 输出结果

// 1,2,3,4,5,6,

filter：数据筛选，相当于if判断

@Test

public void filterTest() {

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

List<Integer> collect = numberList.stream()

.filter(number -> number % 2 == 0)

.collect(Collectors.toList());

collect.forEach(number -> System.out.print(number + ","));

}

distinct：去重

public void distinctTest() {

List<String> list = Arrays.asList("AA", "BB", "CC", "BB", "CC", "AA", "AA");

long l = list.stream().distinct().count();

System.out.println("count:"+l);

String output = list.stream().distinct().collect(Collectors.joining(","));

System.out.println(output);

}

sorted

peek

limit：获取前n个元素

skip：丢弃前n个元素

@Test

public void limitOrSkipTest() {

List<Integer> ageList = Arrays.asList(11, 22, 13, 14, 25, 26);

ageList.stream()

.limit(3)

.forEach(age -> System.out.print(age+","));、//11，22,13

System.out.println();

ageList.stream()

.skip(3)

.forEach(age -> System.out.print(age+","));//14,25,26

}

parallel：并行流

public void parallelTest(){

Long resourse = LongStream.rangeClosed(0,1000000000L)

.parallel().reduce(0,Long::sum);

System.out.println(resourse);

}

sequential

unordered

**聚合收集结果**

stream处理的最后一步，执行完stream就被用尽了不能继续操作。

forEach：遍历stream，不能return/break，支持lambda

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

numberList.stream().forEach(number -> System.out.println(number+","));

forEachOrdered

toArray

reduce：累加器

//reduce中返回的结果会作为下次累加器计算的第一个参数

Optional accResult = Stream.of(1, 2, 3, 4).reduce((acc, item) -> {

System.out.println("acc : " + acc);

acc += item;

System.out.println("item: " + item);

System.out.println("acc+ : " + acc);

System.out.println("--------");

return acc;

});

collect

min

max

count

anyMatch

allMatch

noneMatch

findFirst

findAny

iterator

Statistics：统计

@Test

public void mathTest() {

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);

IntSummaryStatistics stats = list.stream().mapToInt(x -> x).summaryStatistics();

System.out.println("最小值：" + stats.getMin());

System.out.println("最大值：" + stats.getMax());

System.out.println("个数：" + stats.getCount());

System.out.println("和：" + stats.getSum());

System.out.println("平均数：" + stats.getAverage());

}

// 输出结果

// 最小值：1

// 最大值：6

// 个数：6

// 和：21

// 平均数：3.5

groupingBy：分组聚合，相当于mysql的group by

@Test

public void groupByTest() {

List<Integer> ageList = Arrays.asList(11, 22, 13, 14, 25, 26);

Map<String, List<Integer>> ageGrouyByMap = ageList.stream()

.collect(Collectors.groupingBy(age -> String.valueOf(age / 10)));

ageGrouyByMap.forEach((k, v) -> {

System.out.println("年龄" + k + "0多岁的有：" + v);

});

}

// 输出结果

// 年龄10多岁的有：[11, 13, 14]

// 年龄20多岁的有：[22, 25, 26]

partitioningBy：按条件分组

@Test

public void partitioningByTest() {

List<Integer> ageList = Arrays.asList(11, 22, 13, 14, 25, 26);

Map<Boolean, List<Integer>> ageMap = ageList.stream()

.collect(Collectors.partitioningBy(age -> age > 18));

System.out.println("未成年人：" + ageMap.get(false));

System.out.println("成年人：" + ageMap.get(true));

}

// 输出结果

// 未成年人：[11, 13, 14]

// 成年人：[22, 25, 26]

**自己生成Stream**

@Test

public void generateTest(){

// 生成自己的随机数流

Random random = new Random();

Stream<Integer> generateRandom = Stream.generate(random::nextInt);

generateRandom.limit(5).forEach(System.out::println);

// 生成自己的 UUID 流

Stream<UUID> generate = Stream.generate(UUID::randomUUID);

generate.limit(5).forEach(System.out::println);

}

//使用limit进行短路

**short-circuiting**

有一种 Stream 操作被称作 short-circuiting ，它是指当 Stream 流**无限大**但是需要返回的 Stream 流是**有限**的时候，而又希望它能在**有限的时间**内计算出结果，那么这个操作就被称为short-circuiting。例如　findFirst操作。

findFirst：找出stream中第一个元素

@Test

public void findFirstTest(){

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

Optional<Integer> firstNumber = numberList.stream()

.findFirst();

System.out.println(firstNumber.orElse(-1));

}

//找出第一个元素后就会停止遍历，相当于短路操作

解决终端操作只能一个的问题

Supplier<Stream<String>> streamSupplier =

() -> Stream.of("d2", "a2", "b1", "b3", "c")

.filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true); // ok

streamSupplier.get().noneMatch(s -> true); // ok

**并行迭代器**

//tryAdvance 相当于普通迭代器iterator 串行处理

public void iterator(){

AtomicInteger num = new AtomicInteger(0);

while(true){

boolean flag = spliterator.tryAdvance((i) ->{

num.addAndGet((int)i);

System.out.println(i);

});

if(!flag){

break;

}

}

System.out.println(num);

}

//trySplit将list分段，每段单独处理，为并行提供可能

public void spliterator(){

AtomicInteger num = new AtomicInteger(0);

Spliterator s1 = spliterator.trySplit();

Spliterator s2 = spliterator.trySplit();

spliterator.forEachRemaining((i) ->{

num.addAndGet((int)i);

System.out.println("spliterator:"+i);

});

s1.forEachRemaining((i) ->{

num.addAndGet((int)i);

System.out.println("s1:"+i);

});

s2.forEachRemaining((i) ->{

num.addAndGet((int)i);

System.out.println("s2:"+i);

});

System.out.println("最终结果："+num);

}

//利用分段，开启多线程处理

public void spliterator2() throws InterruptedException {

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {

run(spliterator.trySplit());

return "future1 finished!";

});

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {

run(spliterator.trySplit());

return "future2 finished!";

});

CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> {

run(spliterator);

return "future3 finished!";

});

CompletableFuture<Void> combindFuture = CompletableFuture.allOf(future1, future2);

try {

combindFuture.get();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

System.out.println("future1: " + future1.isDone() + " future2: " + future2.isDone());

System.out.println("最终结果为：" + count);

}

public void run(Spliterator s1) {

final String threadName = Thread.currentThread().getName();

System.out.println("线程" + threadName + "开始运行-----");

s1.forEachRemaining(new Consumer() {

@Override

public void accept(Object o) {

count.addAndGet((Integer)o);

}

});

System.out.println("线程" + threadName + "运行结束-----");

}

**HashMap**

JDK8优化了HashMap的实现， 主要优化点包括：

* 将链表方式修改成链表或者红黑树的形式
* 修改resize的过程，解决JDK7在resize在并发场景下死锁的隐患
* JDK1.7存储使用Entry数组，     JDK8使用Node或者TreeNode数组存储

当链表长度大于8是链表的存储结构会被修改成红黑树的形式。

查询效率从O(N)提升到O(logN)。链表长度小于6时，红黑树的方式退化成链表。

JDK7链表插入是从链表头部插入， 在resize的时候会将原来的链表逆序。

JDK8插入从链表尾部插入， 因此在resize的时候仍然保持原来的顺序。

**其他**

新增JVM工具：jdeps提供了用于分析类文件的命令行工具。

使用metaSpace代替永久区

新增NMT(Native     Memeory Trace)本地内存跟踪器

日期和时间api：

老版本的Date类有两个，java.util.Date和java.sql.Date线程不安全