# 【Lambda表达式和函数式编程】

## 主要内容

1. Lambda表达式
2. 函数式接口
3. Stream流式编程
4. 方法引用

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| Lambda表达式 | 掌握 |
| 函数式接口 | 了解 |
| Stream流式编程 | 掌握 |
| 方法引用 | 了解 |

## 一、JDK8新特征

### 1.1 JDK版本变化

|  |  |  |
| --- | --- | --- |
| **JDK版本** | **名称** | **发布时间** |
| 1.0 | Oak(橡树) | 1996/1/23 |
| 1.1 |  | 1997/2/19 |
| 1.2 | Playground（运动场） | 1998/12/4 |
| 1.3 | Kestrel（美洲红隼） | 2000/5/8 |
| 1.4.0 | Merlin（灰背隼） | 2002/2/13 |
| Java SE 5.0 / 1.5 | Tiger（老虎） | 2004/9/30 |
| Java SE 6.0 / 1.6 | Mustang（野马） | 2006/4/1 |
| Java SE 7.0 / 1.7 | Dolphin（海豚） | 2011/7/28 |
| Java SE 8.0 / 1.8 | Spider（蜘蛛） | 2014/3/18 |
| Java SE 9.0 |  | 2017/9/21 |
| Java SE 10.0 |  | 2018/3/21 |
| Java SE 11.0 |  | 2018/9/25 |
| Java SE 12.0 |  | 2019/3/19 |
| Java SE 13.0 |  | 2019/9/17 |
| Java SE 14.0 |  | 2020/3/17 |

2009年4月20日晚，甲骨文和Sun宣布，两家公司已达成正式收购协议。根据协议，甲骨文将以每股9.5美元的价格收购Sun，交易总价值约为74亿美元。所以Java7及其后面的版本都是Oracle公司推出的。

从Java9这个版本开始，Java 的计划发布周期是6个月。这意味着Java的更新从传统的以特性驱动的发布周期，转变为以时间驱动的发布模式，并逐步的将Oracle JDK原商业特性进行开源。

理念：小步快跑，快速迭代。后面周期变短，对之前的东西固定下来保存下来抛弃一些，版本趋于稳定，更新内容新特性就少。使用者就像小白鼠，我们相当于测试了，有一些功能优缺点进行测试和反馈，测试完它就改动，他掌握用户需求就可以改。针对企业客户的需求，Oracle将以三年为周期发布长期支持版本(long term support)。

其中Java5，Java8是革命性的版本。Java11也将提供长期的支持。Java9,10都是“功能性的版本”，支持时间只有半年左右。**特别是JDK8是JDK5以来最具革命性的版本**。主要新特征包括Lambda表达式、函数式接口、方法引用和构造器引用、Stream API、新日期类、接口新变化等。其中的新日期类、接口新变化在前面章节已经讲解。

总的来说,JDK8中的Lambda表达式和Stream 是自Java语言添加泛型(Generics)和注解(annotation)以来最大的变化。

### 1.2 Lambda表达式

#### 1.2.1引入Lambda表达式

#### 【示例1】引入Lambda表达式

|  |
| --- |
| **public interface** MyInterface {  **public void** method(); } **class** MyClass **implements** MyInterface{  **public void** method() {  System.***out***.println(**"使用类实现接口"**);  } } **class** Test{  **public static void** main(String[] args) {  //1.定义类实现接口  MyInterface myInferface1 = **new** MyClass();  myInferface1.method();  //2.使用匿名内部类  MyInterface myInferface2 = **new** MyInterface() {  **public void** method() {  System.***out***.println(**"使用匿名内部类实现类"**);  }  };  myInferface2.method();  //3.使用Lambda表达式  MyInterface myInferface3 = ()->System.***out***.println(**"使用Lambda表达式"**);  myInferface3.method();  } } |

**Lambda表达式：**

Lambda表达式基于数学中的λ演算得名，直接对应于其中的lambda抽象，是一个匿名函数，即没有函数名的函数。

**Lambda表达式好处**：

使用 Lambda 表达式可使代码变的更加简洁紧凑。并且Lambda表达式可和Stream API等相结合，使代码更加简洁紧凑。Lambda 表达式经常用来替代部分匿名内部类。

**Lambda表达式的语法**

(parameters) -> expression或 (parameters) ->{ statements; }

参数：要重写的方法的形参列表

-> ：lambda运算符

表达式/语句体：要重写的方法的方法体

**Lambda表达式的本质：**Lambda 表达式是一种匿名函数（不是匿名内部类），简单地说，它是没有声明的方法，也即没有访问修饰符、返回值声明和名字。它实质属于函数式编程的概念。（不是匿名内部类的，此点在此不做展开）。

#### 1.2.2 使用Lambda表达式

1. **无参数无返回值**

#### 【示例2】无参数无返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface {  **public void** method(); } **class** Test1{  **public static void** main(String[] args) {  MyInterface myInterface = ()->{System.***out***.println(**"无参数无返回值,一条语句"**);};  myInterface.method();  //一条语句：{}可以省略  myInterface = ()->System.***out***.println(**"无参数无返回值,一条语句,省略{}"**);  myInterface.method();  //多条语句：{}不可以省略  /\*final\*/ **int** n = 5;  MyInterface myInferface2 = ()->{  System.***out***.println(**"无参数无返回值"**);  //n = 6;  System.***out***.println(n);  };  myInferface2.method();  } } |

注意：

* 如果Lambda表达式的语句体只有一条语句，{}可以省略；如多条语句，{}不可以省略。
* Lambda 表达式只能引用标记了 final 的外层局部变量。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）。

1. **一个参数无返回值**

#### 【示例3】一个参数无返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface2 {  **public void** method(**int** num1); } **class** Test2{  **public static void** main(String[] args) {  MyInterface2 myInterface = (**int** x)->System.***out***.println(x);  myInterface.method(10);  //类型可以省略  myInterface = (x)->System.***out***.println(x);  myInterface.method(20);  //()可以省略  **myInterface = x->System.*out*.println(x);**  myInterface.method(30);  myInterface = x->{  System.***out***.println(**"-----"**);  System.***out***.println(x);  System.***out***.println(**"-----"**);  };  myInterface.method(40);  } } |

注意：

* Lambda表达式的参数类型可以省略，参数名可以是任意名称；
* 如果只有一个参数,参数外的()也可以省略。

1. **多个参数无返回值**

#### 【示例4】多个参数无返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface3 {  **public void** method(**int** num,String str); } **class** Test3{  **public static void** main(String[] args) {  MyInterface3 myInterface=(**int** num,String str)-> System.***out***.println(num+**" "**+str);  myInterface.method(20,**"bjsxt"**);  //参数的小括号不能省略  myInterface = (num,str)-> System.***out***.println(num+**" "**+str);  myInterface.method(20,**"bjsxt"**);  myInterface = (num,str)-> {  System.***out***.println(**"----------"**);  System.***out***.println(num+**" "**+str);  System.***out***.println(**"----------"**);  };  myInterface.method(20,**"bjsxt"**);  } } |

注意：

* Lambda表达式如果多个参数，参数外的 （）不可省略。

1. **有参数有返回值**

#### 【示例5】有参数有返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface4 {  **public int** method(**int** num); } **class** Test4{  **public static void** main(String[] args) {  MyInterface4 myInterface = (x)-> {**return** x+10; };  **int** result = myInterface.method(10);  System.***out***.println(result);  //只有一条return语句,{}和return可以均省去  **myInterface = (x)-> x+10;**  result = myInterface.method(10);  System.***out***.println(result);  myInterface = (x)-> {  x+=10;  **return** x+10;  };  result = myInterface.method(10);  System.***out***.println(result);  } } |

注意：

* 有返回值的Lambda表达式，如果方法体只有一条语句，可同时省略return和{}。
* 虽然使用 Lambda 表达式可以对某些接口进行简单的实现，但并不是所有的接口都可以使用 Lambda 表达式来实现。**Lambda 规定接口中只能有一个需要被实现的抽象方法，不是规定接口中只能有一个方法，称为函数式接口**。

**总结：**

->左面：

* 参数类型可以省略不写！->类型推断
* 如果只有一个参数，()可以省略不写

->右侧：

* {}将方法体的具体内容包裹起来
* 只有一个方法体执行语句的话，{}可以省略不写
* 如果一句执行语句是return语句，return可以省略不写。

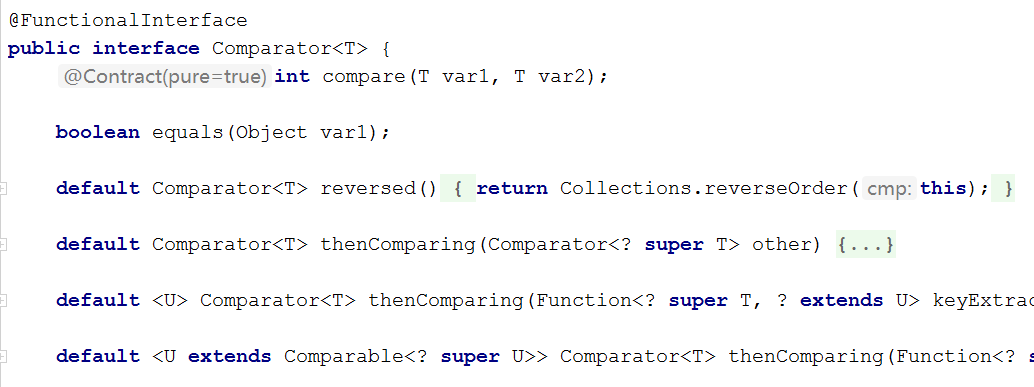
### 1.3函数式接口

#### 1.3.1 认识函数式接口

函数式接口：只能有一个抽象方法，其他的可以有default、static、Object里public方法等。作用：在Java中主要用在Lambda表达式和方法引用（实际上也可认为是Lambda表达式）上。

JDK8专门提供了@FunctionalInterface注解，用来进行编译检查。

已经使用过的函数式接口，比如Comparator等，后面在多线程阶段要学习的函数式接口有Runnable、Callable等。（注意：Comparable并没有被标记为函数式接口）



#### 【示例6】认识函数式接口

|  |
| --- |
| @FunctionalInterface **public interface** FuncInterface {  //只有一个抽象方法  **public void** method1();  //default方法不计  **default void** method2(){  }  //static方法不计  **static void** method3(){  }  //从Object过来的public方法不计  **public boolean** equals(Object obj); } |

#### 1.3.2内置函数式接口。

JDK 也提供了大量的内置函数式接口，使得 Lambda 表达式的运用更加方便、高效。这些内置的函数式接口已经可以解决我们开发过程中绝大部分的问题，只有一小部分比较特殊得情况需要我们自己去定义函数式接口。在这里特别介绍四个函数式接口。

* Consumer<T>：消费型接口（void accept(T t)）。有参数，无返回值
* Supplier<T>：供给型接口（T get（））。只有返回值，没有入参
* Function<T, R>：函数型接口（R apply（T t））。一个输入参数，一个输出参数，两种类型不可不同、可以一致
* Predicate<T>：断言型接口（boolean test（T t））。输入一个参数，输出一个boolean类型得返回值

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数式接口 | 方法名 | 输入参数 | 输出参数 | 作用 |
| 消费型接口  Consumer | void accept(T t) | T | void | 对类型为T的对象进行操作 |
| 供给型接口  Supplier | T get（） | void | T | 返回类型为T的对象 |
| 函数型接口  Function | R apply（T t） | T | R | 对类型为T的对象进行操作，返回类型为R的对象 |
| 断言型接口  Predicate | boolean test（T t） | T | boolean | 对类型为T的对象进行操作，返回布尔类型结果 |

1. **消费型接口Consumer<T>**

#### 【示例7】消费型接口

|  |
| --- |
| **public class** TestFunctional1 {  **public static void** main(String[] args) {  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87);  //使用匿名内部类实现  Consumer consumer = **new** Consumer<Integer>() {  @Override  **public void** accept(Integer elem) {  System.***out***.println(elem);  }  };  list.forEach(consumer);  //使用Lambda表达式  //list.forEach((elem)->{System.out.println(elem);});  list.forEach((elem)->System.***out***.println(elem));  } } |

1. **断言型接口Predicate<T>**

#### 【示例8】断言型接口1

|  |
| --- |
| **public class** TestFunctional2 {  **public static void** main(String[] args) {  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87);  //使用匿名内部类实现  System.***out***.println(list);  Predicate predicate = **new** Predicate<Integer>(){  @Override  **public boolean** test(Integer i) {  **if**(i<60){  **return true**;  }  **return false**;  }  };  list.removeIf(predicate);  System.***out***.println(list);  //使用Lambda表达式实现  list.removeIf((i)->{**if**(i>80) {  **return true**;  }  **return false**;  });  System.***out***.println(list);  } } |

#### 【示例9】断言型接口2

|  |
| --- |
| **public class** TestFunctional3 {  **public static void** main(String[] args) {  List<String > list = **new** ArrayList<>();  Collections.*addAll*(list,**"Java"**,**"MySQL"**,**"HTML"**,**"JSP"**,**"SSM"**);  System.***out***.println(list);   **int** length=4;  Predicate <String>predicate =  (elem)->{**if**(elem.length()>=length ) **return true**; **return false**;};  List<String> list2 = *retailIf*(predicate,list);  System.***out***.println(list2);  }  **public static** List retailIf(Predicate predicate,List<String> list){  List<String> list2 = **new** ArrayList<>();  **for** (String elem :list){  **if**(predicate.test(elem)){  list2.add(elem);  }  }  **return** list2;  } } |

### 1.4方法引用

有时候，Lambda体可能仅调用一个已存在方法，而不做任何其它事，对于这种情况，通过一个方法名字来引用这个已存在的方法会更加清晰。方法引用是一个更加紧凑，易读的 Lambda 表达式，注意方法引用是一个 Lambda 表达式，方法引用操作符是双冒号 "::"。

#### 【示例10】方法引用1

|  |
| --- |
| **public class** TestMethodRef1 {  **public static void** main(String[] args) {  //使用匿名内部类实现  Consumer consumer = **new** Consumer<Integer>() {  @Override  **public void** accept(Integer i) {  System.***out***.println(i);  }  };  consumer.accept(56);  //使用lambda表达式实现  Consumer<Integer> consumer1 = (i)->System.***out***.println(i);  consumer1.accept(56);  //使用方法引用  //println()的参数类型、返回值类型正好和accept方法的参数类型、返回值类型相同  Consumer<Integer> consumer2 = System.***out***::println;  consumer2.accept(56);  } } |

在本示例中，println()的参数类型、返回值类型正好和Consumer接口的accept方法的参数类型、返回值类型相同，此时可以采用方法引用来简化语法。语法为对象名::实例方法名。

#### 【示例11】方法引用2

|  |
| --- |
| **public class** TestMethodRef2 {  **public static void** main(String[] args) {  Student stu = **new** Student(10,**"zhangsan"**,23,100);  //使用匿名内部类实现  Supplier<String> supplier1= **new** Supplier<String>(){  **public** String get() {  **return** stu.getName();  }  };  System.***out***.println(supplier1.get());  //使用Lambda表达式实现  Supplier<String> supplier2 = ()-> stu.getName();  System.***out***.println(supplier2.get());  //使用方法引用实现  Supplier<String> supplier3 = stu::getName;  System.***out***.println(supplier3.get());  } } |

#### 【示例12】方法引用3

|  |
| --- |
| **public class** TestMethodRef3 {  **public static void** main(String[] args) {  //使用匿名内部类实现  Comparator comparator1 = **new** Comparator<Integer>() {  **public int** compare(Integer in1, Integer in2) {  //return in1.intValue()-in2.intValue();  **return** Integer.*compare*(in1,in2);  }  };  System.***out***.println(comparator1.compare(12,34));  //使用Lambda表达式实现  Comparator<Integer> comparator2 = (in1,in2)->{ **return** Integer.*compare*(in1,in2);};  System.***out***.println(comparator2.compare(12,34));  //使用方法引用实现  Comparator<Integer> comparator3 =Integer::*compare*;  System.***out***.println(comparator3.compare(12,34));  } } |

**方法引用有下面几种方式：**

1. 对象引用::实例方法名
2. 类名::静态方法名
3. 类名::实例方法名
4. 类名::new （也称为构造方法引用）
5. 类型[]::new （也称为数组引用）

这里讲解了第一种方式（System.***out***::println、stu::getName）和第二种方式（Integer::*compare*）， 对于另外三种形式，感兴趣的同学可以自己查询资料进行学习。

### 1.5流式编程

Stream作为Java8的一大亮点，它与java.io包里的InputStream和OutputStream是完全不同的概念。它是对容器对象功能的增强，它专注于对容器对象进行各种非常便利、高效的聚合操作或者大批量数据操作。

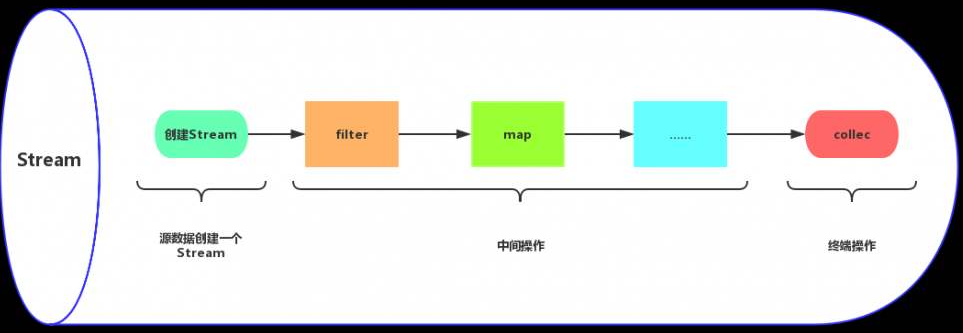
Stream API借助于同样新出现的Lambda表达式，极大的提高编程效率和程序可读性。同时，它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用fork/join并行方式来拆分任务和加速处理过程。所以说，Java8中首次出现的 java.util.stream是一个函数式语言+多核时代综合影响的产物。

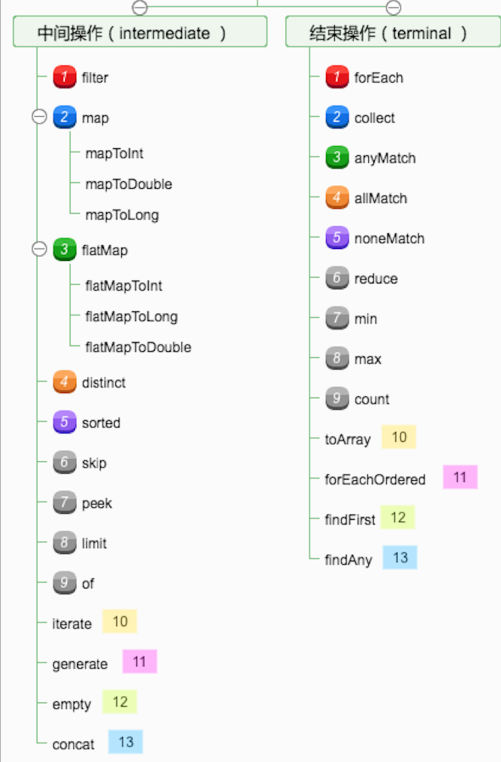
Stream有如下三个操作步骤：

一、创建Stream：从一个数据源，如集合、数组中获取流。

二、中间操作：一个操作的中间链，对数据源的数据进行操作。

三、终止操作：一个终止操作，执行中间操作链，并产生结果。





当数据源中的数据上了流水线后，这个过程对数据进行的所有操作都称为“中间操作”。中间操作仍然会返回一个流对象，因此多个中间操作可以串连起来形成一个流水线。比如map (mapToInt, flatMap 等)、filter、distinct、sorted、peek、limit、skip、parallel、sequential、unordered。

当所有的中间操作完成后，若要将数据从流水线上拿下来，则需要执行终止操作。终止操作将返回一个执行结果，这就是你想要的数据。比如：forEach、forEachOrdered、toArray、reduce、collect、min、max、count、anyMatch、allMatch、noneMatch、findFirst、findAny、iterator。

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何处理！而在终止操作时一次性全部处理，称作“惰性求值”。

#### 【示例13】流式编程1：中间操作和中止操作

|  |
| --- |
| **public class** TestStream1 {  **public static void** main(String[] args) {  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87,80,87,95,100,34,45);  //创建Stream  Stream<Integer> stream = list.stream();  //进行中间操作  stream = stream.filter((x)->{**if**(x>=60) **return true**; **return false**;})//刷选掉不及格的  .distinct() //去重  .sorted((x1,x2)->{**return** -Integer.*compare*(x1,x2);})//降序排列  .limit(4)//只要前四个  .map((x)->x+5)//每个成绩加5分  .skip(2);//跳过前两个，从第三个开始  //进行终止操作:stream has already been operated upon or closed  //stream.forEach(System.out::println);//遍历  //System.out.println(stream.max((x1,x2)->x1-x2));;  //System.out.println(stream.count());  System.***out***.println(stream.findFirst());   } } |

#### 【示例14】流式编程2：并行流和串行流

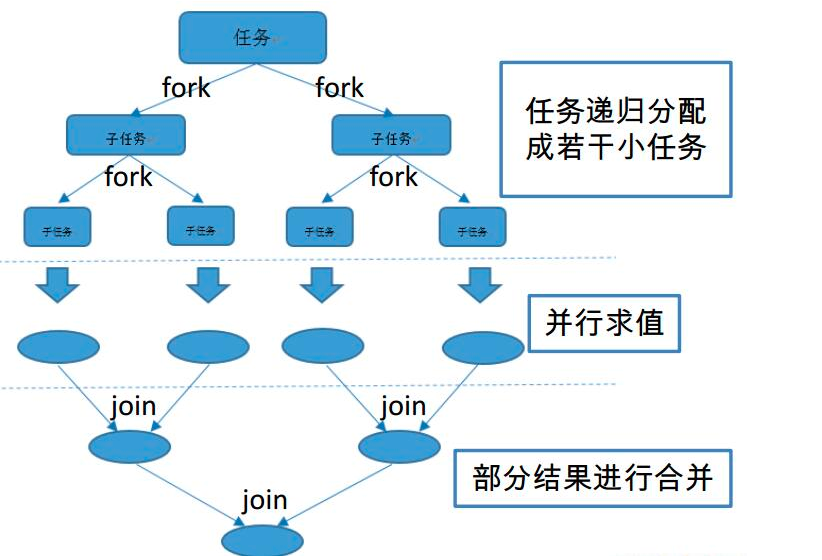
**什么是并行流？**

简单来说，并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。 Stream API 可以声明性地通过 parallel() 与sequential() 在并行流与顺序流之间进行切换 。

**Fork/Join 框架**

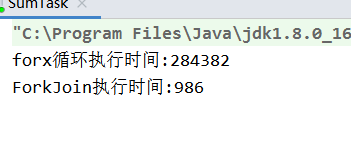
Fork/Join 框架： 就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总 。



**使用ForkJoin和普通for实现1-1000000000000l求和效率对比**

|  |
| --- |
| **package** com.bjsxt.demo;  **import** java.util.concurrent.ExecutionException; **import** java.util.concurrent.ForkJoinPool; **import** java.util.concurrent.Future; **import** java.util.concurrent.RecursiveTask;  **public class** SumTask **extends** RecursiveTask<Long> {   **private long start**;  **private long end**;  **private final int step** = 2000000;*//最小拆分成几个数相加* **public** SumTask(**long** start, **long** end) {  **this**.**start** = start;  **this**.**end** = end;  }  @Override *//在这个方法中定义我们自己计算的规则* **protected** Long compute() {  **long** sum = 0;  **if**(**end** - **start** <= **step** ){  *//System.out.println("小于步长进行求和");  //小于5个数，直接求和* **for** (**long** i = **start**; i <=**end**; i++) {  sum+=i;  }  }**else**{  *//大于5个数，分解任务* **long** mid = (**end** + **start**)/2;  SumTask leftTask = **new** SumTask(**start**,mid);  SumTask rightTask = **new** SumTask(mid+1,**end**);  *//执行子任务* leftTask.fork();  rightTask.fork();  *//子任务，执行完，得到执行结果* **long** leftSum = leftTask.join();  **long** rightSum = rightTask.join();  *//System.out.println("join结果"+leftSum+"---"+rightSum);* sum = leftSum+rightSum;  }  **return** sum;  }   **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  **int** sum=0;  **long** l = System.*currentTimeMillis*();  **for** (**int** i = 0; i <=1000000000000l; i++) {  sum+=i;  }  **long** l2 = System.*currentTimeMillis*();  System.***out***.println(**"forx循环执行时间:"**+(l2-l));  *//使用ForkJoin框架解决  //创建一个ForkJoin池* ForkJoinPool pool = **new** ForkJoinPool();  *//定义一个任务* SumTask sumTask = **new** SumTask(1,1000000000000l);  *//将任务交给线程池* **long** l3 = System.*currentTimeMillis*();  Future<Long> future = pool.submit(sumTask);  *//得到结果并输出* Long result = future.get();  **long** l4 = System.*currentTimeMillis*();   System.***out***.println(**"ForkJoin执行时间:"**+(l4-l3));  } } |

**执行结果：**



|  |
| --- |
| //创建Stream方式2：并行流，底层采用ForkJoin框架，结果并不按照集合原有顺序输出  System.***out***.println(**"----------------"**);  Stream stream2 = list.parallelStream();//  stream2.forEach((x)->System.***out***.println(x+**"---"**+Thread.*currentThread*().getName())); |

#### 【示例15】流式编程3：创建Stream

|  |
| --- |
| **public class** TestStream2 {  **public static void** main(String[] args) {  //创建Stream方式1  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87,80,87,95,100,34,45);  Stream stream = list.stream();  stream.forEach(System.***out***::println);  //创建Stream方式2：并行流，底层采用ForkJoin框架，结果并不按照集合原有顺序输出  System.***out***.println(**"----------------"**);  Stream stream2 = list.parallelStream();//  stream2.forEach((x)->System.***out***.println(x+**"---"**+Thread.*currentThread*().getName()));  //创建Stream方式3:of()  System.***out***.println(**"----------------"**);  Stream stream3 = Stream.*of*(34,56,89,65,87,80,87,95,100,34,45);  stream3.forEach(System.***out***::println);  //创建Stream方式4  **int** [] arr = {34,56,89,65,87,80,87,95,100,34,45};  IntStream stream4 = Arrays.*stream*(arr);  stream4.forEach(System.***out***::println);  //创建Stream方式5  Stream stream5 = Stream.*generate*(()->Math.*random*());  // stream5.forEach(System.out::println);  //创建Stream方式6  Stream stream6 = Stream.*iterate*(6,(i)->2+i);  stream6.limit(5).forEach(System.***out***::println);  } } |

### 本节作业

1. Lambda表达式的语法和本质
2. 函数式接口的特点
3. 四大内置函数式接口
4. 方法引用的5种语法
5. 流式编程的操作步骤