什么是 Lambda 表达式

用 Lambda 表达式用来表示匿名函数，意思是没有函数名字的函数。

()->{} 定义并创建了一个接口的实例。

() 定义**方法参数**。不需要声明参数类型。如果只有一个参数，可以省略()。

{} 定义**方法体**。如果只有一行，则可以省略{}。

Lambda 表达式也叫**函数式编程**，对应命令式编程。

注意，要求该接口只有一个抽象方法。可以在接口定义增加注解@FunctionalInterface，声明该接口是函数式接口。

例子：new Thread(() -> System.out.println("hello"));

等价于:

Runnable runnable = () -> System.out.println("hello");

Thread thread = new Thread(runnable);

显然，由 JDK 判断() -> System.out.println("hello")实现的是 Runnable 接口。

为什么需要 Lambda 表达式

让代码更简洁。Lambda 表达式可以使用非常少的代码实现抽象接口（函数式接口）。

Lambda 表达式是某个函数式接口的具体实现，并且会返回一个函数式接口的对象。

在某些业务场景，接口是固定的，但是实现接口的具体实例只是“临时”的。不需要知道具体实例的类名（所以也叫匿名类），而方法名、参数名和返回值在接口已经定义，也就不需要这个临时的实现类再重复这些代码。

缺点是不如**命令式编程**容易调试。

注意，所有的 Lambda 表达式都可以转换为命令式编程吗？

为什么需要函数式接口

Lambda 表达式调用外部变量

当在某个方法中定义 Lambda 表达式时，Lambda 表达式不可调用该方法的参数或者局部变量。

Lambda 表达式可以调用该方法所在类的成员变量和成员方法。

Lambda 表达式与异常处理

在函数式接口的方法定义需抛出的异常。

在 Lambda 表达式的定义中抛出异常。

在调用 Lambda 表达式的位置处理异常。

Lambda 表达式有哪些常见写法?

(String s, String t) -> {return s.length – t.length;}

左侧()代表方法的参数，右侧{}代表方法的实现。

而整体的() -> {}代表的是一个函数式接口的实现。

其中，如果方法体有多条语句，则用{}。如果只有一条语句，可以省略{}和 return。即 **s.length – t.length;**就是方法的返回值。

(String s, String t) -> s.length – t.length;

如果根据上下文可以推导出方法参数的类型，则可以省略类型的声明。

(s, t) -> s.length – t.length;

如果只有一个方法参数，则可以省略()。

S -> System.out.println(s);

如果没有方法参数，可以直接写()。

() -> System.out.println(“hello”);

()表示对应的函数式接口的方法没有参数。不需要声明返回值的类型，因为总是可以通过上下文推导出。

Lambda 表达式和引用方法

在定义 Lambda 表达式时的语法。

1.引用静态方法

比如类名::静态方法

2.引用成员方法

对象::成员方法

3.引用带泛型的方法

比如类名::<Integer>静态方法

4.引用构造方法

类名::new

函数式接口

指的是有且只有一个抽象方法的接口。

JDK 提供的默认的函数式接口

名称	一元接口	说明	二元接口	说明
一般函数	Function	一元函数，抽象apply方法	BiFunction	二元函数，抽象apply方法
算子函数（输入输出同类型）	UnaryOperator	一元算子，抽象apply方法	BinaryOperator	二元算子，抽象apply方法
谓词函数（输出boolean）	Predicate	一元谓词，抽象test方法	BiPredicate	二元谓词，抽象test方法
消费者（无返回值）	Consumer	一元消费者函数，抽象accept方法	BiConsumer	二元消费者函数，抽象accept方法
供应者（无参数，只有返回值）	Supplier	供应者函数，抽象get方法	-	-

可以根据使用场景，是否需要返回值，返回值类型，入参数量等，选择合适的函数式接口。

例子：

Consumer 接口

只有一个入参，没有返回值

Consumer consumer = s -> System.out.println(s); // 一个入参，无返回值

consumer.accept(“hello”);

等价于：

Consumer consumer = (s) -> {System.out.println(s);}

或 Consumer consumer = (s) -> {System.out::println;}

Supplier 接口

没有入参，只有一个返回值

Supplier<String> supplier = () -> "i am supplier";

log.info(supplier.get());

Function 接口

java.util.function.Function<T,R> 接口用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件.

比如想要进行属性之间的转换，如 String->Integer,就可以使用 Function 接口。Function 的泛型一般有两种类型，**前面的类型 T 表示传入的参数类型，后面的类型 R 表示返回值类型**。

例子：

```
Function<Integer, String> function = s -> s + "123";
System.out.println(function.apply(100));
```

Function 函数接口的特点经常被用于 Stream 流操作过程中，作为 map 方法的入参。

```
String str1 = "Hello My World.";
Stream.of(str1.split(" "))
.filter(s -> s.length() > 2)
.map(s -> s.length())
.forEach(System.out::println);
```

相当于：

```
Function<String, Integer> function = s -> s.length();
Stream.of(str1.split(" "))
.filter(s -> s.length() > 2)
.map(function)
.forEach(System.out::println);
```

注意：以上这些是通用的函数式接口，使用的是泛型。

还有很多专用的函数式接口，比如 Runnable。可以根据使用场景选择。

方法引用

更简洁的写法：

```
Consumer consumer = System.out::println;
```

consumer.accept(“hello”);

如果函数式接口的实现恰好可以通过调用一个方法来实现，那么我们可以使用方法引用。

方法引用又分了几种：

静态方法的方法引用，类名::静态方法名;

非静态方法的方法引用，实例名::非静态方法名;

构造函数的方法引用，类名::new;

最近学到的 Lambda 表达式基础知识_3y-CSDN 博客

Supplier supplier = () -> “hello”; // 无入参，有返回值

String str = supplier.get();

Supplier supplier = MyClass::new; // 把 Lambda 表达式简化为方法调用

String str = supplier.get();

这些默认的函数式接口的使用场景是什么？

java8 Consumer supplier predicate function 简单使用示例_zhaozhen 的博客-CSDN 博客

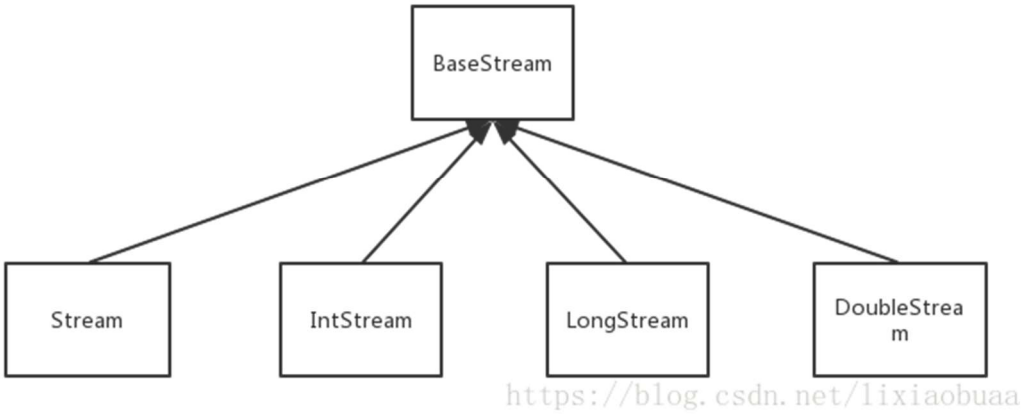
流处理

是 Java 程序中一种重要的数据处理手段，它用少量的代码就可以执行复杂的数据过滤、映射、查找和收集等功能。

流处理的中间操作和最终操作的入参基本都是 **Lambda 表达式**。

缺点是代码的可读性不高。

什么是 Stream 流?



Stream 流不是数据结构，也不保存数据。它是在原数据集上定义了**一组操作**。

因为不保存数据，所以每个 Stream 流只能使用一次。

看源码，本质上是一个接口 interface。

为什么需要 Stream 流?

对于一些常用的针对数组的操作，JDK 库提供了封装好的工具类 API，以 Stream 流的方式。即：数组转换为 Stream 流，调用 Java 提供的默认的 Stream 流的方法，比如求和，去重，过滤等等。

优点是可以调用 JDK 封装好的 API。支持并发。在**多核**情况下，可以使用**并行** Stream API 来发挥多核优势。在单核的情况下，我们自己写的 for 性能不比 Stream API 差多少。

缺点是 Stream 流不好调试。

使用 Stream 流可以更清楚地知道我们需要对数据集做什么操作，**可读性强**。可以轻松获取**并行 Stream 流**，而不用自己编写多线程代码，从而更专注于业务逻辑。

Stream 流和 Lambda 表达式有什么关系?

Stream 流的中间操作和最终操作的入参基本都是 Lambda 表达式。

如何使用 Stream 流?

Java8 的 Stream 流详解_长风-CSDN 博客_java stream 流操作

手把手带你体验 Stream 流_编码博客控的博客-CSDN 博客

使用 Stream 流分为三步:

1）创建 Stream 流;

2）通过 Stream 流对象执行**中间操作**;

3）执行**最终操作**，得到结果;

例子:

```
int[] nums = {1, 2, 3};

int sum = IntStream.of(nums).sum();

log.info("sum: " + sum);
```

1）创建 Stream 流

1.Collection 接口的 stream()或 parallelStream()方法

2.静态的 Stream.of()、Stream.empty()方法

3.Arrays.stream(array, from, to)

4.静态的 Stream.generate()方法生成无限流，接受一个不包含引元的函数

5.静态的 Stream.iterate()方法生成无限流，接受一个种子值以及一个迭代函数

6.Pattern 接口的 splitAsStream(input)方法

7.静态的 Files.lines(path)、Files.lines(path, charSet)方法

8.静态的 Stream.concat()方法将两个流连接起来

例子:

```
List<String> stringList = Arrays.asList("A", "", "B", "abc");
```

stringList.stream(); // 从集合创建，Collection 类提供了 stream 方法

IntStream intStream = **Arrays.*stream*(new int[] {1, 2, 3}); // 从数组创建**

intStream = IntStream.*of*(1, 2, 3); // 直接创建数字流

intStream = new Random().ints().limit(10); // 使用 random 创建

Java8 的 Stream 流详解_长风-CSDN 博客_java stream 流操作

默认情况下，从有序集合、生成器、迭代器产生的 Stream 流，或者通过 Stream.sorted()产生的流都是**有序流。有序流在并行处理完成后都会恢复原顺序。**

unordered()方法可以解除有序流的顺序限制，更好地发挥并行处理的性能优势，例如 distinct 将保存任意一个唯一元素而不是第一个，limit 将保留任意 n 个元素而不是前 n 个。

2）执行中间操作 Intermediate

中间操作是对 Stream 流的数据的加工。**注意，如果不执行最终操作，中间操作不会执行。**

对 Stream 流执行中间操作返回的仍然是 Stream 流。所以可以多个中间操作叠加。

常见的中间操作包括 filter，map，peek 等。

其中 peek 操作主要用于 debug，比如打印中间操作过程的结果。

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

如果数据的类型 T 是 String 类型，对数据进行操作并不会改变最终结果。

比如：Stream.of(“one”, “two”, “three”).peek(s -> s.toUpperCase()).forEach(System.out::println);

而如果是对象类型，则会改变。

这些中间操作和最终操作的参数基本都是 Lambda 表达式。

1.filter(Predicate)

返回一个满足指定条件的流，将结果为 false 的元素过滤掉。

```
String str1 = "Hello My World.";

Stream.of(str1.split(" "))

.filter(s -> s.length() > 2)

.map(s -> s.length())

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

2.map(Function<? Super T, ? extends R> mapper)

转换元素的值，可以用方法引元或者 lambda 表达式。

对流中的元素调用 mapper 方法，产生包含这些元素的一个新的流。

返回值<R> Stream<R>

3.flatMap(function)

若元素是流，将流摊平为正常元素，再进行元素转换。

4.limit(n)

保留前 n 个元素。

5.skip(n)

去除前 n 个元素。

6.distinct()

去除重复元素。

7.sorted()

将 Comparable 元素的流排序。

8.sorted(Comparator)

将流元素按 Comparator 排序。

9.peek(function)

流不变，但会把每个元素传入 fun 执行，可以用作调试。

例子：

```
String str1 = "Hello My World.";

Stream.of(str1.split(" ")).filter(s -> s.length() > 2).map(s -> s.length()).forEach(System.out::println);
```

3）执行最终操作 Terminal

最终操作是对 Stream 流的启动操作。

最终操作返回的不再是 Stream 流对象。

最终操作只能有一个。

约简操作

1. max(Comparator<? Super T> comparator)

根据指定比较器规则，获取流中最大元素。

2.min(Comparator)

3.count()

返回流中元素个数。

4.findFirst()

返回第一个元素。

5.findAny()

返回任意元素。

6.anyMatch(Predicate)

任意元素匹配时返回 true

7.allMatch(Predicate)

所有元素匹配时返回 true。

8.noneMatch(Predicate)

没有元素匹配时返回 true。

9.reduce(fun)

从流中计算某个值，接受一个二元函数作为累积器，从前两个元素开始持续应用它，累积器的中间结果作为第一个参数，流元素作为第二个参数。

10.reduce(a, fun)

a 为幺元值，作为累积器的起点。

11.reduce(a, fun1, fun2)

与二元变形类似，并发操作中，当累积器的第一个参数与第二个参数都为流元素类型时，可以对各个中间结果也应用累积器进行合并，但是当累积器的第一个参数不是流元素类型而是类型 T 的时候，各个中间结果也为类型 T，需要 fun2 来将各个中间结果进行合并。

收集操作

1.iterator()

2.forEach(Consumer<? Super T> action)

遍历流中的每一个元素，执行 action 动作。

顺序执行还是并行执行，取决于流是串行还是并行。

3.forEachOrdered(fun)

可以应用在并行流上以保持元素顺序。

4.toArray()

5.toArray(T[] :: new)

返回正确的元素类型

6.collect(Collector)

7.collect(fun1, fun2, fun3)

fun1 转换流元素；fun2 为累积器，将 fun1 的转换结果累积起来；fun3 为组合器，将并行处理过程中累积器的各个结果组合起来

以上的操作可以分为 3 类

数据过滤：filter(), distinct(), limit(), skip()

数据映射：map()

数据查找：allMatch(), anyMatch(), nonmatch(), findFirst()

问题

Stream 流的这些函数是并行还是串行执行?

乐字节-Java8 核心特性实战之 Stream（流） - 简书 (jianshu.com)

在 Java 8 中，集合接口有两个方法来生成流：

stream() – 为集合创建串行流。

parallelStream() – 为集合创建并行流。

所以看创建的是串行流还是并行流。

性能比较

Stream 并行流详解_我是七月呀的博客-CSDN 博客_stream 并行流

1）基本类型

性能消耗： Stream 串行>for 循环>Stream 并行

2）对象

性能消耗：Stream 串行>for 循环>Stream 并行

3）复杂对象

性能消耗：for 循环>Stream 串行>Stream 并行

Stream 流的这些函数是阻塞的吗?

有阻塞作用，只有当最终命令执行时才会一起链式执行,这得益于 Lamba 的阻塞作用。

Optional 类

是一个容器，可以保存任何对象，并且针对 NullPointerException 空指针异常做了优化，保证 Optional 类保存的值不会是 null。因此 Optional 类是针对“对象可能是 null 也可能不是 null”的场景为开发者提供了优质的解决方案，减少了繁琐的异常处理。

1.empty()

返回一个表示空值的 Optional 实例，即 Optional 实例的成员变量 value=null。

2.filter()

如果 value 有值并且与给定条件匹配，则返回一个包含该值的 Optional 实例。否则返回一个表示空值的 Optional 实例。

3.get()

如果 Optional 实例的 value 是有值的，则返回 value 值，否则抛出异常 NoSuchElementException。

4.of(T value)

要求传入的 value 不能为空，否则抛出 NullPointerException 异常。

5.ofNullable(T value)

允许传入的 value 为空，返回一个 Optional.empty()。

6.orElse(T other)

如果 Optional 实例的 value 是有值的，则返回 value 值，否则返回参数值。

7.isPresent

如果 value 存在（不为空）为 true，否则为 false。

8.ifPresent(Consumer)

如果 Optional 实例有值则为其调用 consumer，否则不做处理。没有返回值。

9.map(Function mapper)

如果 Optional 实例的 value 是有值的，执行 mapper 函数得到返回值。
如果返回值不为空，则对返回值进行封装成 Optional 实例并返回。
否则，返回 Optional.empty()。

10.Flatmap(Function mapper)

和 map()区别在于：要求 mapper 函数返回值的类型为 Optional，并直接返回，不再封装。

使用 Optional 类的意义

嵌套对象 String isocode = user.getAddress().getCountry().getIsocode().toUpperCase();

每一步都可能触发 NullPointerException 异常。

解决：

String ioscode = Optional.ofNullable(user)

.flatmap(user -> user::getAddress)

.flatmap(address -> address::getCountry)

.flatmap(country -> country::getIsocode)

.orElse("default").toUpperCase();

使用场合

不应该用在类的字段，或者是方法的参数。这样让代码复杂而且没有必要。

可以作为方法的返回值

函数式编程

避免使用大量的重复代码来实现司空见惯的功能。

一个**函数**可以作为参数，传递给另外一个函数。

一个函数可以作为返回值。

一个函数可以赋值给一个变量。

在函数式编程中，几乎所有传递的对象都不会被轻易修改。

由于对象处于不变的状态，因此函数式编程更加易于并行。甚至完全不用担心线程安全的问题。

Lambda 表达式是函数式编程和核心。

Lambda 表达式即匿名函数，它是一段没有函数名的函数体，可以作为参数直接传递给相关的调用者。