Lambda表达式

lambda表达式语法

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

函数式接口

函数式接口相关概念

- 函数式接口就是 `只定义一个抽象方法` 的接口，函数式接口 `定义且只定义了一个抽象方法`
- lambda表达式允许直接以内联的形式为函数式接口的抽象方法提供实现，并把整个表达式作为函数式接口的实例

  > e.g：Runnable r1 = () -> System.out.println("Hello World");

- 任何函数式接口都不允许抛出受检异常（checked exception）

  可以在statements里面显示捕捉受检异常

@FunctionalInterface注解

- 如果你用 `@FunctionalInterface` 定义了一个接口，而它却不是函数式接口的话，编译器将返回一个提示原因的错误
- 这个注解不是必须的，就跟 `@Override` 差不多

内置函数式接口

- predicate：谓词
- consumer：消费者
- supplier：生产者
- function：函数
- unaryoperator：一元操作符
- binaryoperator：二元操作符

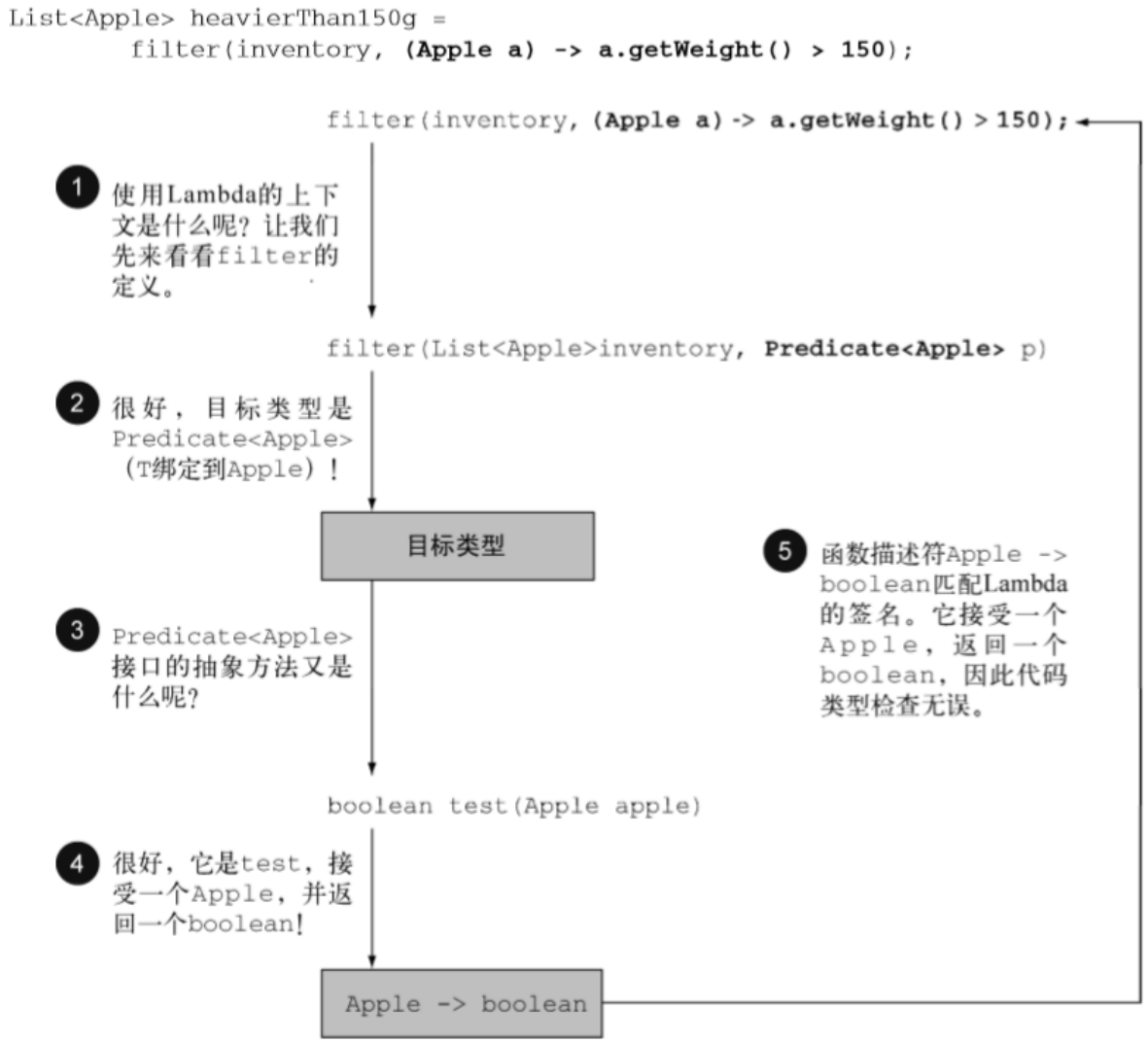
函数式接口	函数描述符
Predicate<T>	T->boolean
Consumer<T>	T->void
Function<T, R>	T->R
Supplier<T>	()->T
UnaryOperator<T>	T->T
BinaryOperator<T>	(T, T)->T
BiPredicate<L, R>	(L, R)->boolean
BiConsumer<T, U>	(T, U)->void
BiFunction<T, U, R>	(T, U)->R

类型检查、类型推断以及限制

- Lambda 表达式本身并不包含它在实现哪个函数式接口的信息，`Lambda的类型是从使用Lambda的上下文推断出来的`
- Java编译器会从上下文（目标类型）推断出用什么函数式接口来配合Lambda表达式，这意味着它也可以推断出适合Lambda的签名，因为函数描述符可以通过目标类型来得到，`编译器可以了解Lambda表达式的参数类型，这样就可以在Lambda语法中省去标注参数类型`

  > e.g：PriorityQueue pq = new PriorityQueue((o1, o2) -> o1-o2);

- 当Lambda仅有一个类型需要推断的参数时，参数名称两边的括号也可以省略



lambda表达式使用局部变量的限制

- Lambda可以没有限制地引用主体中的实例变量和静态变量，**但局部变量必须显式声明为final，或事实上是final**

  > e.g：下面的会编译不了，因为引用的变量不是final的（或者事实上是final的）
  >
  > // Variable used in lambda expression should be final or effectively final
  >
  > int num = 2020;
  >
  > Runnable r = () -> System.out.println(num);
  >
  > num = 2021;

- 实例变量都存储在堆中，而局部变量则保存在栈上。**如果Lambda可以直接访问局部变量，而且Lambda是在一个线程中使用的，则使用Lambda的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量**

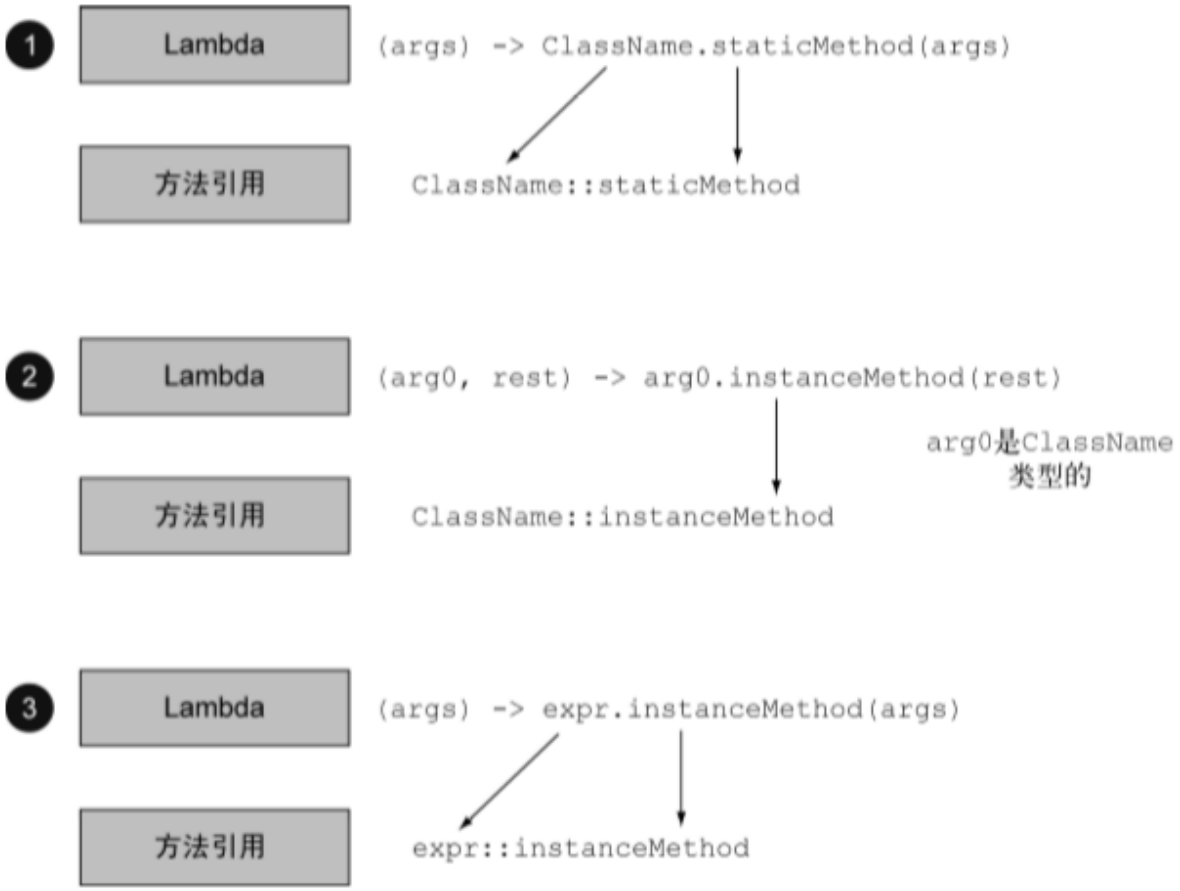
  Java在访问自由局部变量时，实际上是在访问它的副本，而不是访问原始变量

  如果允许捕获可改变的局部变量，就会引发造成线程不安全的新的可能性

方法引用

- 方法引用可以被看作仅仅调用特定方法的Lambda的一种快捷写法
- 针对仅仅涉及单一方法的lambda表达式的语法糖

  lambda表达式仅涉及一个方法的话，可以写成方法引用的样子，看上去简单一点
- 语法：目标引用放在 分隔符`::` 前，方法的名称放在后面



流处理

流的相关概念

流的定义

- 简短的定义就是“从支持数据处理操作的源生成的元素序列”
- 源：集合、数组或输入/输出资源
- 数据处理操作：如filter、map、reduce、find、match、sort等

  流操作可以顺序执行，也可并行执行

- filter——接受Lambda，从流中排除某些元素
- map——接受一个Lambda，将元素转换成其他形式或提取信息
- limit——截断流，使其元素不超过给定数量
- collect——将流转换为其他形式

集合与流的差异

Stream API在决定如何优化这条流水线时更为灵活。例如，筛选、提取和截断操作可以一次进行

- **集合与流之间的差异就在于什么时候进行计算**
- 不管什么时候，集合中的每个元素都是放在内存里的，元素都得先算出来才能成为集合的一部分
- **流就像是一个延迟创建的集合：只有在消费者要求的时候才会计算值**

和迭代器类似，流只能遍历一次。遍历完之后，我们就说这个流已经被消费掉了

外部迭代与内部迭代

- 使用Collection接口需要用户去做迭代（比如用for-each），这称为外部迭代
- Streams库使用内部迭代——它帮你把迭代做了，还把得到的流值存在了某个地方，你只要给出 一个函数说要干什么就可以了

  内部迭代时，项目可以透明地并行处理，或者用更优化的顺序进行处理

  Streams库的内部迭代可以自动选择一种适合你硬件的数据表示和并行实现

流操作

- 中间操作：可以链起来
- 终端操作：最终的操作
- 除非流水线上触发一个终端操作，否则中间操作不会执行任何处理，因为中间操作一般都可以合并起来，在终端操作时一次性全部处理

流的使用

流的常用API

操作	作用	类型	返回类型	函数式接口	函数描述符
filter	过滤	中间	Stream	Predicate	T->boolean
distinct	去重	中间	Stream		
skip	跳过前n个	中间	Stream	long	
limit	只取前n个	中间	Stream	long	
map	对流中每个元素应用函数	中间	Stream	Function<T, R>	T->R
flatmap	扁平化流（合并）	中间	Stream	Function<T, Stream>	T->Stream
sorted	排序	中间	Stream	Comparator	(T, T)->int
anyMatch	任一匹配?	终端	boolean	Predicate	T->boolean
noneMatch	都不匹配?	终端	boolean	Predicate	T->boolean
allMatch	全都匹配?	终端	boolean	Predicate	T->boolean
findAny	任一个	终端	Optional		
findFirst	第一个	终端	Optional		
forEach		终端	void	Consumer	T->void
collect		终端	R	Collector<T, A, R>	
reduce	归约	终端	Optional	BinaryOperator	(T, T)->T
count	计数	终端	long		

原始类型流特化

- 使用普通流的话，对于`Integer、Double、Long`会有暗含的装箱成本，因此引入原始类型特化流接口IntStream、DoubleStream、LongStream

- 使用`mapToInt、mapToDouble、mapToLong`将原始的`Stream<T>`转为`IntStream、DoubleStream、LongStream`
- 使用`boxed`将`IntStream、DoubleStream、LongStream`转回`Stream<T>`
- 数值流常用方法：max、min、average、sum...

  range(a, b)：生成[a, b)范围连续的数字

  rangeClosed(a, b)：生成[a, b]范围连续的数字

流的创建方法

- 静态方法创建流

```
// public static<T> Stream<T> of(T... values)
Stream<T> stream = Stream.of();
```

- 由数组创建流

```
// arr是int[]、double[]、long[]则返回对应的数值流IntStream、DoubleStream、LongStream
// 否则返回Stream<T>
Arrays.stream(arr);
```

- **文件流TODO ch5.7.3**

数据收集

- 传递给`collect方法`的参数是`Collector接口`的一个实现

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

- `Collector`会对元素应用一个转换函数，并将结果累积在一个数据结构中
- `Collectors接口`提供了很多静态工厂方法，可以方便的创建收集器实例

> e.g：collect(Collectors.toList())

归约和汇总，Collectors.reducing工厂方法