链滴

java8 lambda 表达式学习

作者：flowaters

原文链接：https://ld246.com/article/1528369942049

来源网站：链滴

背景

JDK 11已经快要来了，所以JDK的新特征还是要及时学习的。

先从JDK 8的特征来入手吧。

本文先记录一下lambda表达式的学习。

另外，在ClickHouse的学习中，发现ClickHouse也支持了lambda表达式做为SQL的一部分。

所以lambda表达式已经是常识了。

lambda表达式例子

创建线程

```
public class MainLambda {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> System.out.println("Hello world!"));
        thread.start();
    }
}
```

输出

```
Hello world!
```

排序

```
public class MainLambda {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(new String[] { "b", "c", "a" });
        Collections.sort(list, (s1, s2) -> s1.compareTo(s2));
        System.out.println(list);
    }
}
```

输出

```
[a, b, c]
```

转换成大写

```
public class MainLambda {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(new String[] { "b", "c", "a" });
        List<String> newlist = list.stream().map(s -> s.toUpperCase()).collect(Collectors.toList());
        System.out.println(newlist);
    }
}
```

输出

```
[B, C, A]
```

遍历每个元素

```
import java.util.Arrays;
import java.util.List;

public class MainLambda {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(new String[] { "b", "c", "a" });
        list.stream().forEach(s -> System.out.println(s.toUpperCase()));
    }
}
```

输出

```
B
C
A
```

lambda表达式语法

一般语法

```
(Type1 param1, Type2 param2, ..., TypeN paramN) -> {
  statment1;
  statment2;
  //.............
  return statmentM;
}
```

单参数语法

可以省略前面的小括号

```
param1 -> {
  statment1;
  statment2;
  //.............
  return statmentM;
}
```

单语句语法

可以省略后面的大括号, 以及return语句。

```
param1 -> statment
```

方法引用语法

Class or instance :: method

变量作用域

外部变量在lambda表达式引用时，jdk 8 编译器会隐式做为final来处理

stream

上面的转换成小写就是一个stream的例子。

执行概况

生成

collection.stream()

转换

- distinct
- filter
- map
- flatMap
- peak
- limit
- skip

汇聚(Reduce)

collect方法

示例

```
public class MainLambda {
    public static void main(String[] args) {
        List<Integer> nums = Lists.newArrayList(1, 1, null, 2, 3, 4, null, 5, 6, 7, 8, 9, 10);
        List<Integer> numsWithoutNull = nums.stream()
                .filter(num -> num != null)
                .collect(
                        () -> new ArrayList<Integer>(),
                        (list, item) -> list.add(item),
                        (list1, list2) -> list1.addAll(list2)
                );
        System.out.println(numsWithoutNull);
    }
}
```

结果

```
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

其中collect有三个参数

1. supplier: 生成一个新的实例
2. accumulator(对象，元素): 把元素加入对象中
3. combiner(对象, 对象): 合并两个对象

此外，JDK提供了Collector接口，方便来写collect。这里不多写了。

另外，JDK预定义了常用的Collector接口实现，如Collectors.toList()之类。

因此，上面的例子可以写成

```
public class MainLambda {
    public static void main(String[] args) {
        List<Integer> nums = Lists.newArrayList(1, 1, null, 2, 3, 4, null, 5, 6, 7, 8, 9, 10);
        List<Integer> numsWithoutNull = nums.stream()
                .filter(num -> num != null)
                .collect(Collectors.toList());
        System.out.println(numsWithoutNull);
    }
}
```

结果

```
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

reduce方法

示例：求和元素

```
import com.google.common.collect.Lists;

public class MainLambda {
    public static void main(String[] args) {
        List<Integer> ints = Lists.newArrayList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        int value = ints.stream()
                .reduce((sum, item) -> sum + item)
                .get();
        System.out.println("ints sum is:" + value);
    }
}
```

结果

```
ints sum is:55
```

其中reduce的两个参数，第一个参数sum是上一次reduce的返回值，第二个参数是本次的元素。所实际执行过程是：（（（(1+2) + 3) + 4) + 5)

也可以提供一个循环的初始值，如 -1

```
import java.util.List;

import com.google.common.collect.Lists;

public class MainLambda {
    public static void main(String[] args) {
        List<Integer> ints = Lists.newArrayList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        int value = ints.stream()
                .reduce(-1, (sum, item) -> sum + item);
        System.out.println("ints sum is:" + value);
    }
}
```

结果

```
ints sum is:54
```

count

示例

```
import java.util.List;

import com.google.common.collect.Lists;

public class MainLambda {
    public static void main(String[] args) {
        List<Integer> ints = Lists.newArrayList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        long value = ints.stream().count();
        System.out.println("ints sum is:" + value);
    }
}
```

结果

```
ints sum is:10
```

其它

- allMatch: 均匹配
- anyMatch: 有一个匹配
- findFirst: 返回第一个元素，如果为空则返回Optional
- noneMatch: 均不匹配
- max: 最大值
- min: 最小值

执行示例

```
import java.util.List;
```

```
import com.google.common.collect.Lists;

public class MainLambda {
    public static void main(String[] args) {
        List<Integer> nums = Lists.newArrayList(1, 1, null, 2, 3, 4, null, 5, 6, 7, 8, 9, 10);
        int sum = nums.stream()
                .filter(num -> num != null)
                .distinct()
                .mapToInt(num -> num * 2)
                .skip(2)
                .limit(4)
                .peek(System.out::println)
                .sum();
        System.out.println("sum is:" + sum);
    }
}
```

结果: 首先skip掉了前2个元素，又limit了4个元素，所以最后剩下了6,8,10,12共4个元素，其和为36。

```
6
8
10
12
sum is:36
```

并行执行

通过parallelStream，来实现并行执行，默认为8个并发。

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;

import org.apache.http.client.fluent.Request;
import org.apache.http.client.fluent.Response;

public class Java8AsyncRequest {
    public static void main(String[] args) {
        List<String> emitters = new ArrayList<String>();
        for (int i = 0; i != 1000; ++i) {
            emitters.add("http://www.abeffect.com/" + i);

        }

        emitters.parallelStream().map(new Function<String, Response>() {
            @Override
            public Response apply(String s) {
                try {
                    System.out.println(Thread.currentThread() + ": " + s);
                    Thread.sleep(5000L);
```

```
                return Request.Get(s).execute();
            } catch (IOException | InterruptedException e) {
                e.printStackTrace();
                return null;
            }
        }
    }).forEach(new Consumer<Response>() {

        @Override
        public void accept(Response response) {
            try {
                System.out.println(response.returnResponse().getStatusLine().getStatusCode());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
  }
}
```

参考

- Java8特性详解 lambda表达式 Stream
- Java8利用Lambda处理List集合