

06 | Stream如何提高遍历集合效率？

2019-06-01 刘超



你好，我是刘超。

上一讲中，我在讲List集合类，那我想你一定也知道集合的顶端接口Collection。在Java8中，Collection新增了两个流方法，分别是Stream()和parallelStream()。

通过英文名不难猜测，这两个方法肯定和Stream有关，那进一步猜测，是不是和我们熟悉的InputStream和OutputStream也有关系呢？集合类中新增的两个Stream方法到底有什么作用？今天，我们就来深入了解下Stream。

什么是Stream？

现在很多大数据量系统中都存在分表分库的情况。

例如，电商系统中的订单表，常常使用用户ID的Hash值来实现分表分库，这样是为了减少单个表的数据量，优化用户查询订单的速度。

但在后台管理员审核订单时，他们需要将各个数据源的数据查询到应用层之后进行合并操作。

例如，当我们需要查询出过滤条件下的所有订单，并按照订单的某个条件进行排序，单个数据源查询出来的数据是可以按照某个条件进行排序的，但多个数据源查询出来已经排序好的数据，并不代表合并后是正确的排序，所以我们需要在应用层对合并数据集合重新进行排序。

在Java8之前，我们通常是通过for循环或者Iterator迭代来重新排序合并数据，又或者通过重新定

义Collections.sorts的Comparator方法来实现，这两种方式对于大数据量系统来说，效率并不是很理想。

Java8中添加了一个新的接口类Stream，他和我们之前接触的字节流概念不太一样，Java8集合中的Stream相当于高级版的Iterator，他可以通过Lambda 表达式对集合进行各种非常便利、高效的聚合操作（Aggregate Operation），或者大批量数据操作 (Bulk Data Operation)。

Stream的聚合操作与数据库SQL的聚合操作sorted、filter、map等类似。我们在应用层就可以高效地实现类似数据库SQL的聚合操作了，而在数据操作方面，Stream不仅可以通过串行的方式实现数据操作，还可以通过并行的方式处理大批量数据，提高数据的处理效率。

接下来我们就用一个简单的例子来体验下Stream的简洁与强大。

这个Demo的需求是过滤分组一所中学里身高在160cm以上的男女同学，我们先用传统的迭代方式来实现，代码如下：

```
Map<String, List<Student>> stuMap = new HashMap<String, List<Student>>();
for (Student stu: studentsList) {
    if (stu.getHeight() > 160) { //如果身高大于160
        if (stuMap.get(stu.getSex()) == null) { //该性别还没分类
            List<Student> list = new ArrayList<Student>(); //新建该性别学生的列表
            list.add(stu); //将学生放进去列表
            stuMap.put(stu.getSex(), list); //将列表放到map中
        } else { //该性别分类已存在
            stuMap.get(stu.getSex()).add(stu); //该性别分类已存在，则直接放进去即可
        }
    }
}
```

我们再使用Java8中的Stream API进行实现：

1.串行实现

```
Map<String, List<Student>> stuMap = stuList.stream().filter((Student s) -> s.getHeight() > 160).collect(Collectors
```

2.并行实现

```
Map<String, List<Student>>> stuMap = stuList.parallelStream().filter((Student s) -> s.getHeight() > 160).collect(Co
```

通过上面两个简单的例子，我们可以发现，**Stream**结合**Lambda**表达式实现遍历筛选功能非常得简洁和便捷。

Stream如何优化遍历？

上面我们初步了解了**Java8**中的**Stream API**，那**Stream**是如何做到优化迭代的呢？并行又是如何实现的呢？下面我们就透过**Stream**源码剖析**Stream**的实现原理。

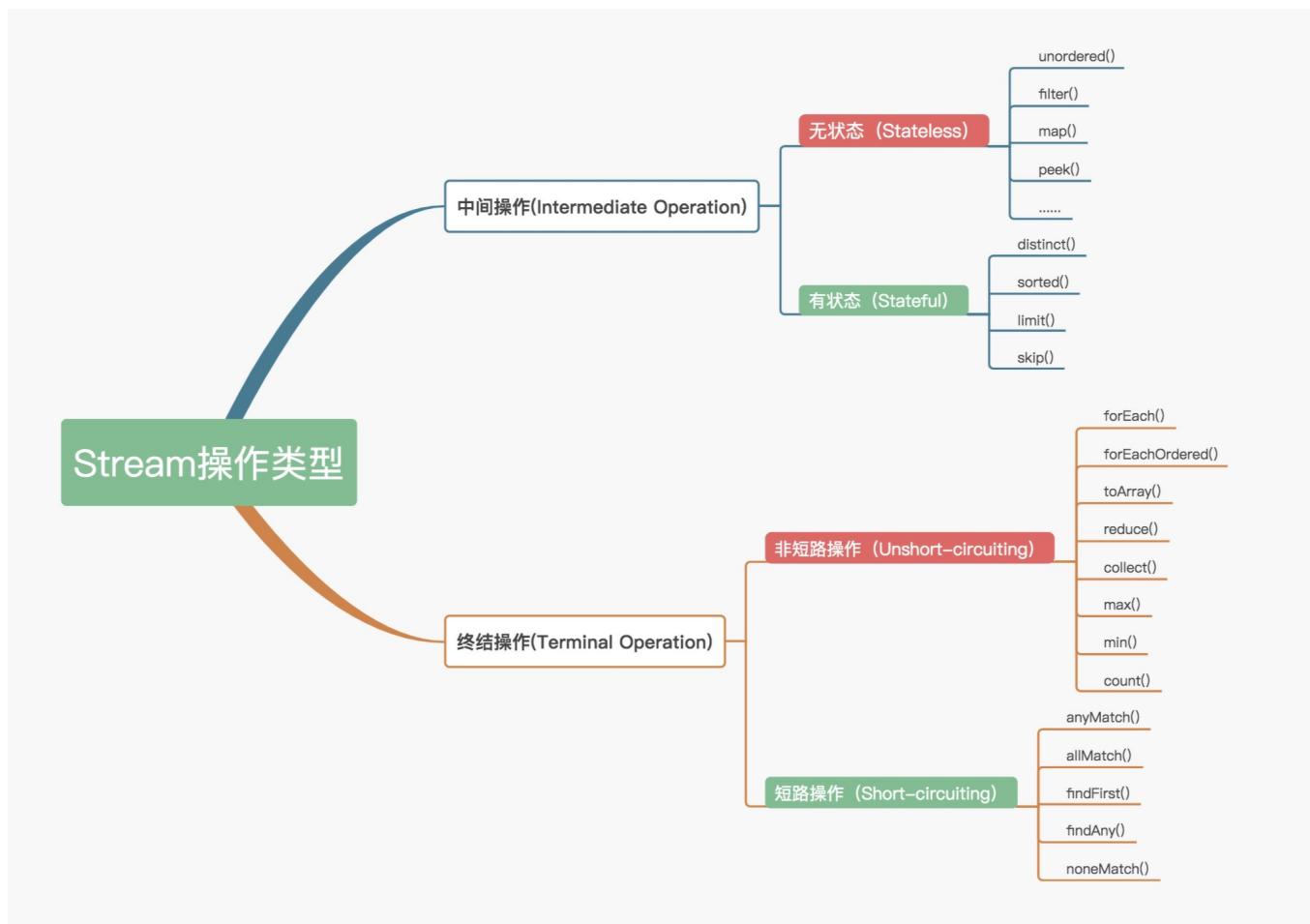
1.Stream操作分类

在了解**Stream**的实现原理之前，我们先来了解下**Stream**的操作分类，因为他的操作分类其实是实现高效迭代大数据集合的重要原因之一。为什么这样说，分析完你就清楚了。

官方将**Stream**中的操作分为两大类：中间操作（**Intermediate operations**）和终结操作（**Terminal operations**）。中间操作只对操作进行了记录，即只会返回一个流，不会进行计算操作，而终结操作是实现了计算操作。

中间操作又可以分为无状态（**Stateless**）与有状态（**Stateful**）操作，前者是指元素的处理不受之前元素的影响，后者是指该操作只有拿到所有元素之后才能继续下去。

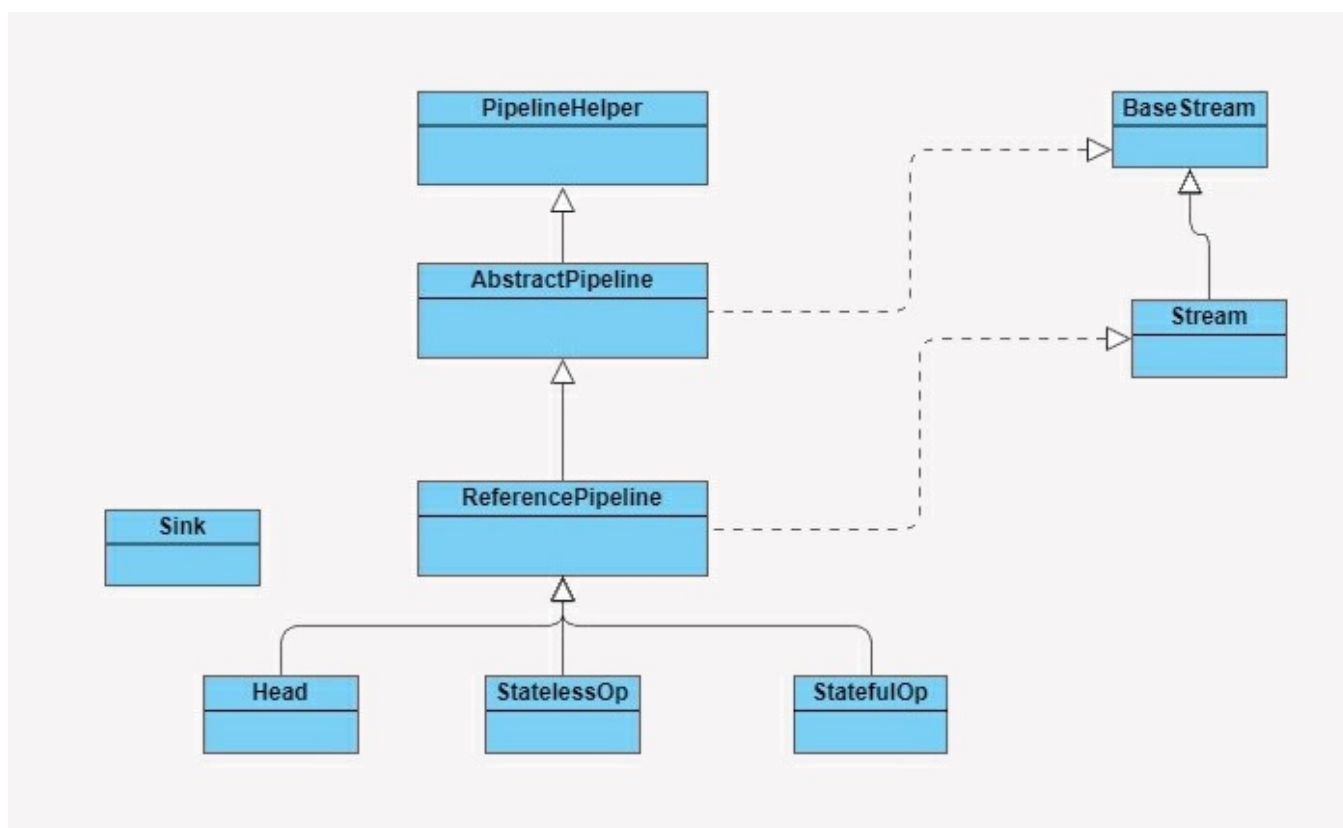
终结操作又可以分为短路（**Short-circuiting**）与非短路（**Unshort-circuiting**）操作，前者是指遇到某些符合条件的元素就可以得到最终结果，后者是指必须处理完所有元素才能得到最终结果。操作分类详情如下图所示：



我们通常还会将中间操作称为懒操作，也正是由这种懒操作结合终结操作、数据源构成的处理管道（**Pipeline**），实现了**Stream**的高效。

2.Stream源码实现

在了解**Stream**如何工作之前，我们先来了解下**Stream**包是由哪些主要结构类组合而成的，各个类的职责是什么。参照下图：



BaseStream和**Stream**为最顶端的接口类。**BaseStream**主要定义了流的基本接口方法，例如，**splitterator**、**isParallel**等；**Stream**则定义了一些流的常用操作方法，例如，**map**、**filter**等。

ReferencePipeline是一个结构类，他通过定义内部类组装了各种操作流。他定义了**Head**、**StatelessOp**、**StatefulOp**三个内部类，实现了**BaseStream**与**Stream**的接口方法。

Sink接口是定义每个**Stream**操作之间关系的协议，他包含**begin()**、**end()**、**cancellationRequested()**、**acctpt()**四个方法。**ReferencePipeline**最终会将整个**Stream**流操作组装成一个调用链，而这条调用链上的各个**Stream**操作的上下关系就是通过**Sink**接口协议来定义实现的。

3.Stream操作叠加

我们知道，一个**Stream**的各个操作是由处理管道组装，并统一完成数据处理的。在JDK中每次的中断操作会以使用阶段（**Stage**）命名。

管道结构通常是由**ReferencePipeline**类实现的，前面讲解**Stream**包结构时，我提到过**ReferencePipeline**包含了**Head**、**StatelessOp**、**StatefulOp**三种内部类。

Head类主要用来定义数据源操作，在我们初次调用**names.stream()**方法时，会初次加载**Head**对象，此时为加载数据源操作；接着加载的是中间操作，分别为无状态中间操作**StatelessOp**对象和有状态操作**StatefulOp**对象，此时的**Stage**并没有执行，而是通过**AbstractPipeline**生成了一个中间操作**Stage**链表；当我们调用终结操作时，会生成一个最终的**Stage**，通过这个**Stage**触发之前的中间操作，从最后一个**Stage**开始，递归产生一个**Sink**链。如下图所示：



下面我们再通过一个例子来感受下**Stream**的操作分类是如何实现高效迭代大数据集合的。

```
List<String> names = Arrays.asList("张三", "李四", "王老五", "李三", "刘老四", "王小二", "张四", "张五六七");

String maxLenStartWithZ = names.stream()
    .filter(name -> name.startsWith("张"))
    .mapToInt(String::length)
    .max()
    .toString();
```

这个例子的需求是查找出一个长度最长，并且以张为姓氏的名字。从代码角度来看，你可能会认为是这样的操作流程：首先遍历一次集合，得到以“张”开头的名字；然后遍历一次**filter**得到的集合，将名字转换成数字长度；最后再从长度集合中找到最长的那个名字并且返回。

这里我要很明确地告诉你，实际情况并非如此。我们来逐步分析下这个方法里所有的操作是如何执行的。

首先，因为**names**是**ArrayList**集合，所以**names.stream()**方法将会调用集合类基础接口**Collection**的**Stream**方法：

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

然后，**Stream**方法就会调用**StreamSupport**类的**Stream**方法，方法中初始化了一个**ReferencePipeline**的**Head**内部类对象：

```

public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel) {
    Objects.requireNonNull(spliterator);
    return new ReferencePipeline.Head<>(spliterator,
                                        StreamOpFlag.fromCharacteristics(spliterator),
                                        parallel);
}

```

再调用**filter**和**map**方法，这两个方法都是无状态的中间操作，所以执行**filter**和**map**操作时，并没有进行任何的操作，而是分别创建了一个**Stage**来标识用户的每一次操作。

而通常情况下**Stream**的操作又需要一个回调函数，所以一个完整的**Stage**是由数据来源、操作、回调函数组成的三元组来表示。如下图所示，分别是**ReferencePipeline**的**filter**方法和**map**方法：

```

@Override
public final Stream<P_OUT> filter(Predicate<? super P_OUT> predicate) {
    Objects.requireNonNull(predicate);
    return new StatelessOp<P_OUT, P_OUT>(this, StreamShape.REFERENCE,
                                        StreamOpFlag.NOT_SIZED) {

        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<P_OUT> sink) {
            return new Sink.ChainedReference<P_OUT, P_OUT>(sink) {

                @Override
                public void begin(long size) {
                    downstream.begin(-1);
                }

                @Override
                public void accept(P_OUT u) {
                    if (predicate.test(u))
                        downstream.accept(u);
                }
            };
        }
    };
}

```

```

@Override
@SuppressWarnings("unchecked")
public final <R> Stream<R> map(Function<? super P_OUT, ? extends R> mapper) {
    Objects.requireNonNull(mapper);
    return new StatelessOp<P_OUT, R>(this, StreamShape.REFERENCE,
        StreamOpFlag.NOT_SORTED | StreamOpFlag.NOT_DISTINCT) {

        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<R> sink) {
            return new Sink.ChainedReference<P_OUT, R>(sink) {

                @Override
                public void accept(P_OUT u) {
                    downstream.accept(mapper.apply(u));
                }

            };
        }
    };
}

```

new StatelessOp将会调用父类**AbstractPipeline**的构造函数，这个构造函数将前后的**Stage**联系起来，生成一个**Stage**链表：

```

AbstractPipeline(AbstractPipeline<?, E_IN, ?> previousStage, int opFlags) {
    if (previousStage.linkedOrConsumed)
        throw new IllegalStateException(MSG_STREAM_LINKED);
    previousStage.linkedOrConsumed = true;
    previousStage.nextStage = this; //将当前的stage的next指针指向之前的stage

    this.previousStage = previousStage; //赋值当前stage当全局变量previousStage
    this.sourceOrOpFlags = opFlags & StreamOpFlag.OP_MASK;
    this.combinedFlags = StreamOpFlag.combineOpFlags(opFlags, previousStage.combinedFlags);
    this.sourceStage = previousStage.sourceStage;
    if (opIsStateful())
        sourceStage.sourceAnyStateful = true;
    this.depth = previousStage.depth + 1;
}

```


因为在创建每一个**Stage**时，都会包含一个**opWrapSink()**方法，该方法会把一个操作的具体实现封装在**Sink**类中，**Sink**采用（处理->转发）的模式来叠加操作。

当执行**max**方法时，会调用**ReferencePipeline**的**max**方法，此时由于**max**方法是终结操作，所以会创建一个**TerminalOp**操作，同时创建一个**ReducingSink**，并且将操作封装在**Sink**类中。

```
@Override
public final Optional<P_OUT> max(Comparator<? super P_OUT> comparator) {
    return reduce(BinaryOperator.maxBy(comparator));
}
```

最后，调用**AbstractPipeline**的**wrapSink**方法，该方法会调用**opWrapSink**生成一个**Sink**链表，**Sink**链表中的每一个**Sink**都封装了一个操作的具体实现。

```
@Override
@SuppressWarnings("unchecked")
final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {
    Objects.requireNonNull(sink);

    for ( @SuppressWarnings("rawtypes") AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p=p.previousStage)
        sink = p.opWrapSink(p.previousStage.combinedFlags, sink);
    }
    return (Sink<P_IN>) sink;
}
```

当**Sink**链表生成完成后，**Stream**开始执行，通过**spliterator**迭代集合，执行**Sink**链表中的具体操作。

```

@Override
final <P_IN> void copyInto(Sink<P_IN> wrappedSink, Spliterator<P_IN> spliterator) {
    Objects.requireNonNull(wrappedSink);

    if (!StreamOpFlag.SHORT_CIRCUIT.isKnown(getStreamAndOpFlags())) {
        wrappedSink.begin(spliterator.getExactSizeIfKnown());
        spliterator.forEachRemaining(wrappedSink);
        wrappedSink.end();
    }
    else {
        copyIntoWithCancel(wrappedSink, spliterator);
    }
}

```

Java8中的Spliterator的forEachRemaining会迭代集合，每迭代一次，都会执行一次filter操作，如果filter操作通过，就会触发map操作，然后将结果放入到临时数组object中，再进行下一次的迭代。完成中间操作后，就会触发终结操作max。

这就是串行处理方式了，那么Stream的另一种处理数据的方式又是怎么操作的呢？

4.Stream并行处理

Stream处理数据的方式有两种，串行处理和并行处理。要实现并行处理，我们只需要在例子的代码中新增一个Parallel()方法，代码如下所示：

```

List<String> names = Arrays.asList("张三", "李四", "王老五", "李三", "刘老四", "王小二", "张四", "张五六七");

String maxLenStartWithZ = names.stream()
    .parallel()
    .filter(name -> name.startsWith("张"))
    .mapToInt(String::length)
    .max()
    .toString();

```

Stream的并行处理在执行终结操作之前，跟串行处理的实现是一样的。而在调用终结方法之后，实现的方式就有点不太一样，会调用TerminalOp的evaluateParallel方法进行并行处理。

```

final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
    assert getOutputShape() == terminalOp.inputShape();
    if (linkedOrConsumed)
        throw new IllegalStateException(MSG_STREAM_LINKED);
    linkedOrConsumed = true;

    return isParallel()
        ? terminalOp.evaluateParallel(this, sourceSpliterator(terminalOp.getOpFlags()))
        : terminalOp.evaluateSequential(this, sourceSpliterator(terminalOp.getOpFlags()));
}

```

这里的并行处理指的是，**Stream**结合了**ForkJoin**框架，对**Stream**处理进行了分片，**Splititerator**中的**estimateSize**方法会估算出分片的数据量。

ForkJoin框架和估算算法，在这里我就不具体讲解了，如果感兴趣，你可以深入源码分析下该算法的实现。

通过预估的数据量获取最小处理单元的阈值，如果当前分片大小大于最小处理单元的阈值，就继续切分集合。每个分片将会生成一个**Sink**链表，当所有的分片操作完成后，**ForkJoin**框架将会合并分片任何结果集。

合理使用Stream

看到这里，你应该对**Stream API**是如何优化集合遍历有个清晰的认知了。**Stream API**用起来简洁，还能并行处理，那是不是使用**Stream API**，系统性能就更好呢？通过一组测试，我们一探究竟。

我们将对常规的迭代、**Stream**串行迭代以及**Stream**并行迭代进行性能测试对比，迭代循环中，我们将对数据进行过滤、分组等操作。分别进行以下几组测试：

- 多核CPU服务器配置环境下，对比长度100的int数组的性能；
- 多核CPU服务器配置环境下，对比长度1.00E+8的int数组的性能；
- 多核CPU服务器配置环境下，对比长度1.00E+8对象数组过滤分组的性能；
- 单核CPU服务器配置环境下，对比长度1.00E+8对象数组过滤分组的性能。

由于篇幅有限，我这里直接给出统计结果，你也可以自己去验证一下，具体的测试代码可以在[Github](#)上查看。通过以上测试，我统计出的测试结果如下（迭代使用时间）：

- 常规的迭代<**Stream**并行迭代<**Stream**串行迭代
- **Stream**并行迭代<常规的迭代<**Stream**串行迭代

- **Stream**并行迭代<常规的迭代<**Stream**串行迭代
- 常规的迭代<**Stream**串行迭代<**Stream**并行迭代

通过以上测试结果，我们可以看到：在循环迭代次数较少的情况下，常规的迭代方式性能反而更好；在单核CPU服务器配置环境中，也是常规迭代方式更有优势；而在大数据循环迭代中，如果服务器是多核CPU的情况下，**Stream**的并行迭代优势明显。所以我们在平时处理大数据的集合时，应该尽量考虑将应用部署在多核CPU环境下，并且使用**Stream**的并行迭代方式进行处理。

用事实说话，我们看到其实使用**Stream**未必可以使系统性能更佳，还是要结合应用场景进行选择，也就是合理地使用**Stream**。

总结

纵观**Stream**的设计实现，非常值得我们学习。从大的设计方向上来说，**Stream**将整个操作分解为了链式结构，不仅简化了遍历操作，还为实现了并行计算打下了基础。

从小的分类方向上来说，**Stream**将遍历元素的操作和对元素的计算分为中间操作和终结操作，而中间操作又根据元素之间状态有无干扰分为有状态和无状态操作，实现了链结构中的不同阶段。

在串行处理操作中，**Stream**在执行每一步中间操作时，并不会做实际的数据操作处理，而是将这些中间操作串联起来，最终由终结操作触发，生成一个数据处理链表，通过Java8中的**Spliterator**迭代器进行数据处理；此时，每执行一次迭代，就对所有的无状态的中间操作进行数据处理，而对有状态的中间操作，就需要迭代处理完所有的数据，再进行处理操作；最后就是进行终结操作的数据处理。

在并行处理操作中，**Stream**对中间操作基本跟串行处理方式是一样的，但在终结操作中，**Stream**将结合**ForkJoin**框架对集合进行切片处理，**ForkJoin**框架将每个切片的处理结果Join合并起来。最后就是要注意**Stream**的使用场景。

思考题

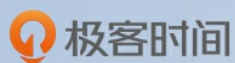
这里有一个简单的并行处理案例，请你找出其中存在的问题。

```
//使用一个容器装载100个数字，通过Stream并行处理的方式将容器中为单数的数字转移到容器parallelList
List<Integer> integerList= new ArrayList<Integer>();

for (int i = 0; i <100; i++) {
    integerList.add(i);
}

List<Integer> parallelList = new ArrayList<Integer>() ;
integerList.stream()
    .parallel()
    .filter(i->i%2==1)
    .forEach(i->parallelList.add(i));
```

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。



Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言



小白猪

5



思考题，由于流是并行处理，`parallelList`会存在并发问题，应该使用`collect`方法聚合

2019-06-01



Loubobooo

👍 4

`parallelList`集合里呈现的是无序的数字，是这样吗？

2019-06-01

作者回复

对的，可能会出现少数字、无序以及异常情况

2019-06-02



(‘田ω田’)

👍 4

感觉这一节课已经值回了整个课程的票价，给老师点赞！

思考题：**Stream**并行执行，无法确认每个元素的处理顺序，最后`parallelList`中的数字是无序的

2019-06-01

作者回复

思考题中的问题是在并行操作`arraylist`时，需要考虑线程安全问题

2019-06-01



Darren

👍 2

最终的结果无序，且可能结果都是不正确的，因为`ArrayList`是线程不安全的

2019-06-04



QQ怪

👍 2

是不是该把思考题中的`arraylist`换成线程安全的`copyOnwriteList`就可以解决线程不安全问题？

2019-06-03

作者回复

对的，但`copyOnwriteList`更适合某一时间段统一新增，且新增时避免大量操作容器发生。比较适合在深夜更新黑名单类似的业务。

2019-06-04



小辉辉

👍 2

`ArrayList`是线程不安全的集合，而当前又用了并行流去处理，所以会出现有异常、少数据或者正常输出结果这三种情况。

2019-06-02



一路看风景

👍 2

老师您好，在容器盛行的微服务环境下，以及大数据处理流行的潮流中，我觉得**stream**的应用空间多少有些尴尬呢，不知是不是我的理解有误。即：单核容器运行的环境下**stream**没了性能优势，大数据的处理又有大数据平台去完成使命，所以是不是意味着我们可以从**stream**得到的最大收益变成了流式编程和函数式编程带来的代码易读和易用性了呢？

2019-06-01

作者回复

是的，但未必所有公司都有构建大数据的能力，而且一些公司有自己的中间件团队，例如文章开始说到的分表分库的查询操作，使用stream的并行操作就有优势了

2019-06-01



bro.

👍 2

老师，这么早更新，读完感觉跟rxjava设计思想很接近，不订阅前面过滤条件都不会真正的运行！

2019-06-01



阿厚

👍 1

老师，请教2个问题：

1.有什么分表分库中间件推荐么？

2.分表分库以后，查询分页怎么办呢？

2019-06-06

作者回复

之前用过sharing-jdbc以及mycat，一个明显的区别是sharing-jdbc是嵌入方式，而mycat是基于proxy，所以理论上来说 proxy方式会有性能损耗。现在我们在使用sharing-jdbc，这里不打广告，两个中间件都有自己的优势。

分页查询是基于我这篇文章说的方式，将每个分表的数据结果集查询出来，通过归并排序计算出。

具体的实现方式有区别，本次专栏的后面课程也会具体讲到。

2019-06-07



郑晨Cc

👍 1

课程好值啊 全是干货

2019-06-03



Liam

👍 1

并发操作一个ArrayList，会有线程安全问题？

2019-06-01

作者回复

对的

2019-06-01



👍

👍 0

首先是测试结果，无序的问题是其一，不过也不算问题。另一个问题是，有可能会最后结果只有47~49个值的现象（实际值应该为50个）。并且多次循环的话会报下标越界。

自认为是ArrayList的并发问题。

2019-06-10

作者回复

对了！

2019-06-10



迎风劲草

👍 0

老师，为什么`stream`操作，就比自己循环的效率高呢，没看懂。

2019-06-05

作者回复

这里强调的是使用`stream`的并发处理大数据时，效率高于传统的遍历处理。

2019-06-06



乐

👍 0

那请问老师，思考题中如何解决这种并发时的线程安全问题？是使用 `CopyOnWriteArrayList` 还是使用 `.collect(Collectors.toList())`？

2019-06-05

作者回复

两者都可以，不过这里如果要使用线程安全集合，可以使用`vector`。

2019-06-06



假装自己不胖

👍 0

例子中查询长度最长并且以张为姓氏的名字,如果有两个会怎么样

2019-06-04

作者回复

结果是一样的，算出最大值。可以复制代码实践下，注意复制后的空格符问题。

2019-06-05



程序员人生

👍 0

是因为并行处理并且`List`是非线程安全的缘故吗？那段代码执行几次后会出现`null`，把`parallel`去掉就好了

2019-06-03

作者回复

是的，但如果需要并行计算时，我们又怎么来处理这类问题呢？

2019-06-03



程序员人生

👍 0

`ArrayList`不是线程安全的，而`parallel()`又是并行流，是不是会有问题？

2019-06-03



fl

👍 0

可以具体解释下`splititerator`吗

2019-06-01



a、

👍 0

运行老师的那段代码,发现会提示`ArrayIndexOutOfBoundsException`,想到`arraylist`是非线程安全的，于是就把`parallelList`改成了`Vector`，运行多次，并未发现异常。然后看到有学员说用`collect`方法，我就改成`conllect`方法，也没有出现异常。对比了两个方法，发现用`verctor`比用`collect`方法的性能要高，但是`collect`方法出来的`list`是排好序的，而`Vector`是乱序的，于是我把数据量调

到了1千万，**Vector**再加上排序，发现也比**collect**方法要快。不是很清楚，为什么并行的处理大数据量也比加锁的要慢？

2019-06-01



我知道了嗯

👍 0

思考题结果是无序的并且有**null**值？这是为什么

2019-06-01

作者回复

ArrayList不是线程安全的，在并行操作时，会出现多线程操作问题，例如出现**null**值，有可能是在扩容时，复制出现问题。同时也会出现值被覆盖的情况。

2019-06-02