모듈 java.base

패키지 java.util.stream

컬렉션에 대한 맵 축소 변환과 같은 요소 스트림에 대한 기능 스타일 작업을 지원하는 클래스입니다. 예를 들면 :

```
int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

여기 widgets에서 Collection<Widget>, a를 스트림의 소스로 사용하고 스트림에 대해 필터 맵 축소를 수행하여 빨간색 위젯의 가중치 합계를 얻습니다. (합산은 축소 작업 의 예입니다 .)

이 패키지에 소개 된 핵심 추상화는 stream 입니다. 클래스 Stream, IntStream, LongStream및 DoubleStream 목적 및 기본 스트림들 위에있는 int, long및 double종류. 스트림은 여러 가지면에서 컬렉션과 다릅니다.

- 저장 공간이 없습니다. 스트림은 요소를 저장하는 데이터 구조가 아닙니다. 대신 데이터 구조, 배열, 생성기 함수 또는 I / O 채널과 같은 소스에서 계산 작업 파이프 라인을 통해 요소를 전달합니다.
- 본질적으로 기능적입니다. 스트림에 대한 작업은 결과를 생성하지만 소스를 수정하지는 않습니다. 예를 들어 Stream 컬렉션에서 가져온 항목을 필터링 Stream하면 원본 컬렉션에서 요소를 제거하는 대신 필터링 된 요소없이 새 항목이 생성 됩니다.
- 게으름 추구. 필터링, 매핑 또는 중복 제거와 같은 많은 스트림 작업을 느리게 구현하여 최적화 기회를 제공 할 수 있습니다. 예를 들어, " String3 개의 연속 모음이 있는 첫 번째 모음 찾기 "는 모든 입력 문자열을 검사 할 필요가 없습니다. 스트림 작업은 중간 (- Stream생성) 작업과 최종 (가치 또는 부작용 생성) 작업 으로 나뉩니다 . 중간 작업은 항상 지연됩니다.
- 제한이 없을 수 있습니다. 컬렉션의 크기는 한정되어 있지만 스트림은 그럴 필요가 없습니다. limit(n)또는 과 같은 단락 연산 findFirst()을 사용하면 무한 스트림에 대한 계산이 유한 시간 내에 완료 될 수 있습니다.
- 소모품. 스트림의 요소는 스트림의 수명 동안 한 번만 방문됩니다. 와 마찬가지로 Iterator소스의 동일한 요소를 다시 방문하려면 새 스트림을 생성해야합니다.

스트림은 여러 가지 방법으로 얻을 수 있습니다. 몇 가지 예는 다음과 같습니다.

- A로부터 Collection비아 stream()와 parallelStream()방법;
- 배열에서 Arrays.stream(Object[]);
- 스트림 클래스에 대한 정적 팩토리 메소드 (예 : Stream.of(Object[]), IntStream.range(int, int) 또는 Stream.iterate(Object, UnaryOperator);
- 파일의 행은 다음에서 얻을 수 있습니다 BufferedReader.lines().
- 파일 경로 스트림은의 메소드에서 얻을 수 있습니다 Files.
- 난수의 스트림은 다음에서 얻을 수 있습니다 Random.ints().
- JDK의 수많은 다른 스트림 베어링 방법을 포함 BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence)및 JarFile.stream().

이러한 기술을 사용하여 타사 라이브러리에서 추가 스트림 소스를 제공 할 수 있습니다 .

스트림 작업 및 파이프 라인

스트림 작업은 중간 작업 과 터미널 작업 으로 나뉘며 결합되어 스트림 파이프 라인 을 형성 합니다 . 스트림 파이프 라인은 소스 (예 : `Collection`, 배열, 생성기 함수 또는 I / O 채널)로 구성됩니다. 0 개 이상의 중간 연산 (예 : `Stream.filter`또는 `Stream.map`; 및 단말 조작 등 `Stream.forEach`이나 `Stream.reduce`.

중간 작업은 새 스트림을 반환합니다. 그들은 항상 게으르다 . 과 같은 중간 작업을 실행하면 `filter()`실제로 필터링을 수행하지 않지만 대신 순회 할 때 주어진 술어와 일치하는 초기 스트림의 요소를 포함하는 새 스트림을 생성합니다. 파이프 라인 소스의 순회는 파이프 라인의 터미널 작업이 실행될 때까지 시작되지 않습니다.

같은 터미널 작업 `Stream.forEach`하거나 `IntStream.sum`, 결과 또는 부작용을 생성하기 위해 상기 스트림을 통과 할 수있다. 터미널 작업이 수행 된 후 스트림 파이프 라인은 소비 된 것으로 간주되어 더 이상 사용할 수 없습니다. 동일한 데이터 소스를 다시 탐색해야하는 경우 데이터 소스로 돌아가서 새 스트림을 가져와야합니다. 거의 모든 경우에 터미널 작업은 열성적 이며 반환하기 전에 데이터 소스의 순회 및 파이프 라인 처리를 완료합니다. 터미널 만 동작 `iterator()`하고 `spliterator()`있지 않으며, 이는 기존 작업이 작업에 충분하지 않은 경우 임의의 클라이언트 제어 파이프 라인 순회를 가능하게하는 "탈출 해치"로 제공됩니다.

스트림을 느리게 처리하면 상당한 효율성을 얻을 수 있습니다. 위의 filter-map-sum 예제와 같은 파이프 라인에서 필터링, 매핑 및 합산은 최소한의 중간 상태로 데이터에 대한 단일 패스로 융합 될 수 있습니다. 게으름은 또한 필요하지 않을 때 모든 데이터를 검사하지 않도록합니다. "1000 자보다 긴 첫 번째 문자열 찾기"와 같은 작업의 경우 소스에서 사용할 수있는 모든 문자열을 검사하지 않고 원하는 특성을 가진 문자열을 찾기 위해 충분한 문자열 만 검사하면됩니다. (이 동작은 입력 스트림이 단순히 크지 않고 무한 할 때 더욱 중요해집니다.)

중간 작업은 상태 비 저장 및 상태 저장 작업 으로 더 나뉩니다 . 이러한 무국적 연산 `filter` 하고 `map`새로운 요소를 처리 할 때, 이전에 본 소자로부터의 상태를 유지하지 - 각각의 요소는 다른 요소에 독립적으로 동작에 처리 될 수있다. 같은 조작 상태, `distinct`및 `sorted`새로운 요소를 처리 할 때, 이전에 본 소자로부터 상태를 포함 할 수있다.

상태 저장 작업은 결과를 생성하기 전에 전체 입력을 처리해야 할 수 있습니다. 예를 들어, 스트림의 모든 요소를 볼 때까지 스트림 정렬 결과를 생성 할 수 없습니다. 결과적으로 병렬 계산에서 상태 저장 중간 작업을 포함하는 일부 파이프 라인은 데이터에 대한 다중 패스가 필요하거나 중요한 데이터를 버퍼링해야 할 수 있습니다. 배타적으로 상태 비 저장 중간 작업을 포함하는 파이프 라인은 데이터 버퍼링을 최소화하면서 순차 또는 병렬에 관계없이 단일 패스로 처리 할 수 있습니다.

또한 일부 작업은 단락 작업으로 간주 됩니다. 무한 입력이 제공 될 때 결과적으로 유한 스트림을 생성 할 수있는 경우 중간 작업이 단락됩니다. 무한 입력이 제공 될 때 유한 시간 내에 종료 될 수 있는 경우 터미널 작동이 단락 된 것입니다. 파이프 라인에서 단락 작업을하는 것은 무한 스트림 처리가 유한 한 시간 내에 정상적으로 종료되기위한 필수 조건이지만 충분하지는 않습니다.

병행

명시 적 `for`-루프가 있는 처리 요소 는 본질적으로 직렬입니다. 스트림은 계산을 각 개별 요소에 대한 명령 적 작업이 아닌 집계 작업의 파이프 라인으로 재구성하여 병렬 실행을 용이하게합니다. 모든 스트림 작업은 직렬 또는 병렬로 실행할 수 있습니다. 병렬 처리가 명시 적으로 요청되지 않는 한 JDK의 스트림 구현은 직렬 스트림을 만듭니다. 예를 들어, `Collection`메소드 보유 `Collection.stream()`하고 `Collection.parallelStream()`순차 생성하고 각각 병렬 스트림;

IntStream.range(int, int) 순차 스트림 을 생성 하는 것과 같은 다른 스트림 을 포함하는 BaseStream.parallel()방법은 해당 방법 을 호출하여 효율적으로 병렬화 할 수 있습니다 . 이전의 "위젯 가중치 합계"쿼리를 병렬로 실행하려면 다음을 수행합니다.

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

이 예제의 직렬 버전과 병렬 버전의 유일한 차이점은 " parallelStream()"대신 " "를 사용하여 초기 스트림을 만드는 것입니다 stream(). 스트림 파이프 라인은 터미널 작업이 호출되는 스트림의 모드에 따라 순차적으로 또는 병렬로 실행됩니다. 스트림의 순차 또는 병렬 모드는 BaseStream.isParallel()메서드 로 결정할 수 있으며 스트림의 모드는 BaseStream.sequential()및 BaseStream.parallel()작업 으로 수정할 수 있습니다 . 가장 최근의 순차 또는 병렬 모드 설정은 전체 스트림 파이프 라인의 실행에 적용됩니다.

findAny()스트림이 순차적으로 실행되는지 병렬로 실행되는지 여부 와 같이 명시 적으로 비 결정적인 것으로 식별 된 작업을 제외하고 는 계산 결과를 변경해서는 안됩니다.

대부분의 스트림 작업은 종종 람다 식인 사용자 지정 동작을 설명하는 매개 변수를 허용합니다. 올바른 동작을 유지하려면 이러한 동작 매개 변수 가 간섭하지 않아야하며 대부분의 경우 상태 비 저장 이어야합니다 . 이러한 매개 변수는 항상 과 같은 기능적 인터페이스의 인스턴스이며, Function종종 람다 식 또는 메서드 참조입니다.

불간섭

Streams를 사용하면 .NET과 같은 스레드로부터 안전하지 않은 컬렉션을 포함하여 다양한 데이터 소스에 대해 병렬 집계 작업을 실행할 수 있습니다 ArrayList. 이는 스트림 파이프 라인 실행 중에 데이터 소스와의 간섭 을 방지 할 수있는 경우에만 가능합니다 . 이스케이프 해치 동작을 제외 iterator()하고 spliterator()실행 단말 동작이 호출 될 때 시작하고, 끝 단말 동작 완료. 대부분의 데이터 소스에서 간섭 방지는 데이터 소스가 전혀 수정되지 않도록하는 것을 의미 합니다.스트림 파이프 라인 실행 중. 이에 대한 주목할만한 예외는 소스가 동시 수정을 처리하도록 특별히 설계된 동시 컬렉션 인 스트림입니다. 동시 스트림 소스는 특성을 Spliterator보고하는 소스 CONCURRENT입니다.

따라서 소스가 동시 적이 지 않을 수있는 스트림 파이프 라인의 동작 매개 변수는 스트림의 데이터 소스를 수정해서는 안됩니다. 동작 매개 변수는 스트림의 데이터 소스를 수정하거나 수정하게 되면 비 동시 데이터 소스 를 방해 한다고합니다 . 비 간섭의 필요성은 병렬 파이프 라인뿐만 아니라 모든 파이프 라인에 적용됩니다. 스트림 소스가 동시 적이 지 않은 경우 스트림 파이프 라인 실행 중에 스트림의 데이터 소스를 수정하면 예외, 오답 또는 부적합 동작이 발생할 수 있습니다. 잘 작동하는 스트림 소스의 경우 터미널 작동이 시작되기 전에 소스를 수정할 수 있으며 이러한 수정 사항은 해당 요소에 반영됩니다. 예를 들어, 다음 코드를 고려하십시오.

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));
Stream<String> sl = l.stream();
l.add("three");
String s = sl.collect(joining(" "));
```

먼저 두 개의 문자열로 구성된 목록이 생성됩니다. "one"; 그리고 "2". 그런 다음 해당 목록에서 스트림이 생성됩니다. 다음으로 세 번째 문자열 "three"를 추가하여 목록을 수정합니다. 마지막으로 스트림의 요소가 수집되고 결합됩니다. 터미널 collect 작업이 시작 되기 전에 목록이 수정 되었으

므로 결과는 "one two three"문자열이됩니다. JDK 컬렉션과 대부분의 다른 JDK 클래스에서 반환 된 모든 스트림은 이러한 방식으로 잘 작동합니다. 다른 라이브러리에서 생성 된 스트림의 경우 잘 작동하는 스트림을 빌드하기위한 요구 사항은 저수준 스트림 구성 을 참조하세요 .

상태 비 저장 동작

스트림 작업에 대한 동작 매개 변수가 상태 저장 인 경우 스트림 파이프 라인 결과는 비 결정적이거나 올바르지 않을 수 있습니다 . 상태 저장 람다 (또는 적절한 기능 인터페이스를 구현하는 다른 객체)는 스트림 파이프 라인 실행 중에 변경 될 수있는 상태에 따라 결과가 달라지는 것입니다. 상태 저장 람다의 예는 `map()`in에 대한 매개 변수 입니다.

```
Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());
stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })...
```

여기서 매핑 작업이 병렬로 수행되면 스레드 스케줄링 차이로 인해 동일한 입력에 대한 결과가 실행마다 다를 수 있지만 상태 비 저장 람다 식을 사용하면 결과가 항상 동일합니다.

또한 동작 매개 변수에서 변경 가능한 상태에 액세스하려고하면 안전 및 성능 측면에서 잘못된 선택을 할 수 있습니다. 해당 상태에 대한 액세스를 동기화하지 않으면 데이터 경합이 발생하여 코드가 손상되지만 해당 상태에 대한 액세스를 동기화하면 경합으로 인해 이익을 얻으려는 병렬 처리가 손상 될 위험이 있습니다. 가장 좋은 방법은 상태 저장 동작 매개 변수를 피하여 작업을 완전히 스트리밍하는 것입니다. 일반적으로 상태 저장을 피하기 위해 스트림 파이프 라인을 재구성하는 방법이 있습니다.

부작용

스트림 작업에 대한 동작 매개 변수의 부작용은 일반적으로 권장되지 않습니다. 이는 무의식적으로 상태 비 저장 요구 사항 및 기타 스레드 안전 위험을 초래할 수 있기 때문입니다.

행동 매개 변수에 부작용이있는 경우, 명시 적으로 언급하지 않는 한 다음과 같은 보장이 없습니다.

- 다른 스레드에 대한 이러한 부작용 의 가시성 ;
- 동일한 스트림 파이프 라인 내의 "동일한"요소에 대한 다른 작업은 동일한 스레드에서 실행됩니다. 과
- 동작 매개 변수는 항상 호출됩니다. 왜냐하면 스트림 구현이 계산 결과에 영향을 미치지 않을 것이라는 것을 증명할 수 있다면 스트림 파이프 라인에서 작업 (또는 전체 단계)을 자유롭게 제거 할 수 있기 때문입니다.

부작용의 순서는 놀랍습니다. 파이프 라인이 스트림 소스의 발생 순서와 일치 하는 결과 를 생성하도록 제한되어있는 경우에도 (예 : `IntStream.range(0,5).parallel().map(x -> x*2).toArray()` must generate `[0, 2, 4, 6, 8]`) 매퍼 함수가 개별 요소에 적용되는 순서에 대해 보장되지 않습니다. 주어진 요소에 대해 어떤 동작 매개 변수가 실행되는 스레드.

부작용을 제거하는 것도 놀랍습니다. 터미널 작업 및를 제외 `forEach`하고 `forEachOrdered`, 스트림 구현이 계산 결과에 영향을주지 않고 동작 매개 변수의 실행을 최적화 할 수있을 때 동작 매개 변수의 부작용이 항상 실행되는 것은 아닙니다. (구체적인 예는 `count` 작업 에 대해 설명 된 API 노트를 참조하십시오 .)

부작용을 사용하려는 유혹을받을 수있는 많은 계산 은 가변 누산기 대신 감소 를 사용하는 것과 같이 부작용없이 더 안전하고 효율적으로 표현 될 수 있습니다 . 그러나 `println()`디버깅 목적 으로

사용 하는 것과 같은 부작용 은 일반적으로 무해합니다. 스트림 연산 같은 소수 forEach()와 peek() 만 부작용을 통해 동작 할 수있다; 주의해서 사용해야합니다.

부적절하게 부작용을 사용하는 스트림 파이프 라인을 그렇지 않은 파이프 라인으로 변환하는 방법의 예로서 다음 코드는 문자열 스트림에서 지정된 정규식과 일치하는 항목을 검색하고 일치 항목을 목록에 넣습니다.

```
ArrayList<String> results = new ArrayList<>();
stream.filter(s -> pattern.matcher(s).matches())
      .forEach(s -> results.add(s));  // Unnecessary use of side-effects!
```

이 코드는 불필요하게 부작용을 사용합니다. 병렬로 실행하면의 비 스레드 안전성으로 ArrayList인해 잘못된 결과가 발생하고 필요한 동기화를 추가하면 경합이 발생하여 병렬 처리의 이점이 손상됩니다. 또한 여기에서 부작용을 사용하는 것은 완전히 불필요합니다. 은 forEach()단순히 병렬로 더욱 안전하고 효율적으로, 그리고 더 많은 의무가 감소 동작으로 대체 할 수있다 :

```
List<String>results =
    stream.filter(s -> pattern.matcher(s).matches())
          .collect(Collectors.toList());  // No side-effects!
```

주문

스트림에는 정의 된 발생 순서 가있을 수도 있고 없을 수도 있습니다 . 스트림에 발생 순서가 있는지 여부는 소스 및 중간 작업에 따라 다릅니다. 특정 스트림 소스 (예 : List또는 배열)는 본질적으로 정렬 된 반면 다른 스트림 소스 (예 :) HashSet는 그렇지 않습니다. 와 같은 일부 중간 작업 sorted()은 순서가 지정되지 않은 스트림에 만남 순서를 부과 할 수 있으며 다른 작업은 순서가 지정된 스트림을 순서가 지정되지 않은 스트림으로 렌더링 할 수 있습니다 BaseStream.unordered(). 또한 일부 터미널 작업은 forEach().

스트림이 정렬 된 경우 대부분의 작업은 발생 순서대로 요소에서 작동하도록 제한됩니다. 스트림의 소스가 List 포함 [1, 2, 3]하는 경우 실행 결과는 map(x -> x*2) 이어야합니다 [2, 4, 6]. 그러나 소스에 정의 된 발생 순서가 없으면 값의 모든 순열이 [2, 4, 6]유효한 결과가됩니다.

순차 스트림의 경우 발생 순서의 유무는 성능에 영향을주지 않고 결정성에 만 영향을줍니다. 스트림이 정렬 된 경우 동일한 소스에서 동일한 스트림 파이프 라인을 반복 실행하면 동일한 결과가 생성됩니다. 순서가 지정되지 않은 경우 반복 실행하면 다른 결과가 생성 될 수 있습니다.

병렬 스트림의 경우 순서 제한을 완화하면 때때로 더 효율적인 실행이 가능할 수 있습니다. 중복 필터링 (distinct()) 또는 그룹화 된 축소 (Collectors.groupingBy()) 와 같은 특정 집계 작업 은 요소 순서가 관련이없는 경우보다 효율적으로 구현할 수 있습니다. 마찬가지로,와 같이 발생 순서에 본질적으로 연결된 작업은 limit()적절한 순서를 보장하기 위해 버퍼링이 필요할 수 있으며, 이는 병렬 처리의 이점을 약화시킵니다. 스트림에 만남 순서가 있지만 사용자가 그 만남 순서에 특별히 신경 쓰지 않는 경우 , 명시 적으로 스트림 순서를 취소합니다.unordered()일부 상태 저장 또는 터미널 작업의 병렬 성능을 향상시킬 수 있습니다. 그러나 위의 "블록 가중치 합계"예제와 같은 대부분의 스트림 파이프 라인은 순서 제약 조건에서도 여전히 효율적으로 병렬화됩니다.

감소 작업

감소 (도 불리는 동작 배량) 같은리스트에 숫자 또는 집적 소자들의 세트의 합 또는 최대 값을 찾는 등의 결합 동작의 반복 적용에 의해 단일 요약 결과에 입력 요소와 콤바인들을 시퀀스를 취 .

스트림 클래스라는 일반적인 감소 작업의 다양한 형태가 reduce() 하고 collect(),뿐만 아니라 여러 전문 감소의 형태 sum(), max()또는를 count().

물론 이러한 작업은 다음과 같이 간단한 순차 루프로 쉽게 구현할 수 있습니다.

```
int sum = 0;
for (int x : numbers) {
   sum += x;
}
```

그러나 위와 같은 돌연변이 축적보다 감소 작업을 선호하는 데에는 좋은 이유가 있습니다. 감소는 "더 추상적"일뿐만 아니라 개별 요소가 아닌 스트림 전체에서 작동 할뿐만 아니라 요소를 처리하는 데 사용되는 함수가 연관 되어있는 한 적절하게 구성된 감소 작업은 본질적으로 병렬화 할 수 있습니다. 및 무 상태 . 예를 들어, 합계를 구하려는 숫자의 흐름이 주어지면 다음과 같이 작성할 수 있습니다.

```
int sum = numbers.stream().reduce(0, (x,y) -> x+y);
```

또는:

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

이러한 축소 작업은 거의 수정없이 안전하게 병렬로 실행할 수 있습니다.

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

구현이 데이터의 하위 집합에 대해 병렬로 작동 한 다음 중간 결과를 결합하여 최종 정답을 얻을 수 있기 때문에 축소가 잘 병렬화됩니다. (언어가 "parallel for-each"구조를 가지고 있더라도, mutative accumulation 접근법은 여전히 개발자가 공유 된 accumulating 변수에 스레드로부터 안전한 업데이트를 제공 sum해야했고, 필요한 동기화는 병렬 처리로 인한 성능 향상을 제거 할 것입니다.) reduce()대신 사용 하면 축소 작업을 병렬화하는 모든 부담이 제거되며 라이브러리는 추가 동기화없이 효율적인 병렬 구현을 제공 할 수 있습니다.

The "widgets" examples shown earlier shows how reduction combines with other operations to replace for loops with bulk operations. If `widgets` is a collection of `Widget` objects, which have a `getWeight` method, we can find the heaviest widget with:

```
OptionalInt heaviest = widgets.parallelStream()
                              .mapToInt(Widget::getWeight)
                              .max();
```

In its more general form, a `reduce` operation on elements of type `<T>` yielding a result of type `<U>` requires three parameters:

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner);
```

여기서 identity 요소는 축소에 대한 초기 시드 값과 입력 요소가없는 경우 기본 결과입니다. 누산기 함수 부분 결과, 다음 요소를 취하고, 새로운 부분적인 결과를 생성한다. 결합기 기능은 새로운 부분 결과를 생성하기 위해 두 부분 결과를 결합한다. (컴 바이 너는 입력이 분할되고 각 분할에 대해 부분 누적이 계산 된 다음 부분 결과가 결합되어 최종 결과를 생성하는 병렬 감소에 필요합니다.)

More formally, the `identity` value must be an identity for the combiner function. This means that for all `u`, `combiner.apply(identity, u)` is equal to `u`. Additionally, the `combiner` function must be associative and must be compatible with the `accumulator` function: for all `u` and `t`, `combiner.apply(u, accumulator.apply(identity, t))` must be `equals()` to `accumulator.apply(u, t)`.

The three-argument form is a generalization of the two-argument form, incorporating a mapping step into the accumulation step. We could re-cast the simple sum-of-weights example using the more general form as follows:

```
int sumOfWeights = widgets.stream()
                          .reduce(0,
                                  (sum, b) -> sum + b.getWeight(),
                                  Integer::sum);
```

명시적인 map-reduce 형식이 더 읽기 쉬우므로 일반적으로 선호됩니다. 일반화 된 형식은 매핑과 축소를 단일 함수로 결합하여 중요한 작업을 최적화 할 수있는 경우에 제공됩니다.

가변 감소

가변 감소 동작은 같은 결과로서 가변 용기에 입력 요소를 축적 `Collection`또는 `StringBuilder`이 스트림의 요소가 처리.

문자열 스트림을 가져 와서 하나의 긴 문자열로 연결하려면 일반적인 감소를 통해이를 달성 할 수 있습니다 .

```
String concatenated = strings.reduce("", String::concat)
```

We would get the desired result, and it would even work in parallel. However, we might not be happy about the performance! Such an implementation would do a great deal of string copying, and the run time would be O(n^2) in the number of characters. A more performant approach would be to accumulate the results into a `StringBuilder`, which is a mutable container for accumulating strings. We can use the same technique to parallelize mutable reduction as we do with ordinary reduction.

The mutable reduction operation is called `collect()`, as it collects together the desired results into a result container such as a `Collection`. A `collect` operation requires three functions: a supplier function to construct new instances of the result container, an accumulator function to incorporate an input element into a result container, and a combining function to merge the contents of one result container into another. The form of this is very similar to the general form of ordinary reduction:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

와 마찬가지로이 추상적 인 방식으로 `reduce()`표현하는 것의 이점은 `collect`병렬화에 직접 적용 할 수 있다는 것입니다. 누적 및 결합 함수가 적절한 요구 사항을 충족하는 한 부분 결과를 병렬로 누적 한 다음 결합 할 수 있습니다. 예를 들어 스트림에있는 요소의 문자열 표현을으로 수집 `ArrayList`하려면 명확한 순차적 for-each 형식을 작성할 수 있습니다.

```
ArrayList<String> strings = new ArrayList<>();
for (T element : stream) {
    strings.add(element.toString());
}
```

또는 병렬화 가능한 수집 양식을 사용할 수 있습니다.

```
ArrayList<String> strings = stream.collect(() -> new ArrayList<>(),
                                           (c, e) -> c.add(e.toString()),
                                           (c1, c2) -> c1.addAll(c2));
```

또는 누산기 함수에서 매핑 연산을 꺼내면 다음과 같이 더 간결하게 표현할 수 있습니다.

```
List<String> strings = stream.map(Object::toString)
                             .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

여기서 우리의 공급자는 단지 `ArrayList constructor`이고 누산기는 문자열 화 된 요소를에 추가 `ArrayList`하고 결합기는 단순히 `addAll`한 컨테이너에서 다른 컨테이너로 문자열을 복사하는 데 사용 합니다.

The three aspects of `collect` -- supplier, accumulator, and combiner -- are tightly coupled. We can use the abstraction of a `Collector` to capture all three aspects. The above example for collecting strings into a `List` can be rewritten using a standard `Collector` as:

```
List<String> strings = stream.map(Object::toString)
                             .collect(Collectors.toList());
```

Packaging mutable reductions into a Collector has another advantage: composability. The class `Collectors` contains a number of predefined factories for collectors, including combinators that transform one collector into another. For example, suppose we have a collector that computes the sum of the salaries of a stream of employees, as follows:

```
Collector<Employee, ?, Integer> summingSalaries
    = Collectors.summingInt(Employee::getSalary);
```

(`?`두 번째 유형 매개 변수는이 수집가에서 사용하는 중간 표현에 관심이 없음을 나타냅니다.) 부서별 급여 합계를 표로 만들기 위해 수집기를 만들려면 다음을 `summingSalaries`사용하여 재사용 할 수 있습니다 `groupingBy`.

```
Map<Department, Integer> salariesByDept
    = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,
                                                       summingSalaries));
```

일반 축소 작업과 마찬가지로 collect()적절한 조건이 충족되는 경우에만 작업을 병렬화 할 수 있습니다. 부분적으로 누적 된 결과의 경우 빈 결과 컨테이너와 결합하면 동등한 결과가 생성되어야 합니다. 즉, p일련의 누산기 및 결합기 호출의 결과 인 부분적으로 누적 된 결과 의 경우와 p동일해야합니다 combiner.apply(p, supplier.get()).

또한 계산이 분할되지만 동일한 결과를 생성해야합니다. 입력 요소 t1및 t2의 경우 결과 r1와 r2아래 계산 의 결과 가 동일해야합니다.

```
A a1 = supplier.get();
accumulator.accept(a1, t1);
accumulator.accept(a1, t2);
R r1 = finisher.apply(a1);  // result without splitting

A a2 = supplier.get();
accumulator.accept(a2, t1);
A a3 = supplier.get();
accumulator.accept(a3, t2);
R r2 = finisher.apply(combiner.apply(a2, a3));  // result with splitting
```

여기서 동등성은 일반적으로에 따라 의미합니다 Object.equals(Object). 그러나 어떤 경우에는 동등성이 완화되어 순서의 차이를 설명 할 수 있습니다.

감소, 동시성 및 순서 지정

예를 들어 다음 collect()과 같은를 생성하는 a Map와 같은 복잡한 축소 작업이 있습니다.

```
Map<Buyer, List<Transaction>> salesByBuyer
    = txns.parallelStream()
          .collect(Collectors.groupingBy(Transaction::getBuyer));
```

병렬로 작업을 수행하는 것은 실제로 비생산적 일 수 있습니다. Map일부 Map구현 에서는 결합 단계 (키로 서로 병합)가 비용이 많이들 수 있기 때문 입니다.

Suppose, however, that the result container used in this reduction was a concurrently modifiable collection -- such as a ConcurrentHashMap. In that case, the parallel invocations of the accumulator could actually deposit their results concurrently into the same shared result container, eliminating the need for the combiner to merge distinct result containers. This potentially provides a boost to the parallel execution performance. We call this a concurrent reduction.

Collector동시 감소를 지원 하는 A 는 Collector.Characteristics.CONCURRENT 특성으로 표시됩니다 . 그러나 동시 수집에는 단점도 있습니다. 여러 스레드가 결과를 공유 컨테이너에 동시에 배치하는 경우 결과가 배치되는 순서는 결정적이지 않습니다. 결과적으로 동시 감소는 처리중인 스트림에 대해 순서가 중요하지 않은 경우에만 가능합니다. Stream.collect(Collector) 구현은 동시 감소하는 경우를 수행합니다

- 스트림은 병렬입니다.
- 수집가는 Collector.Characteristics.CONCURRENT특성을 가지고 있으며;
- 스트림이 순서가 지정되지 않았거나 수집기에 Collector.Characteristics.UNORDERED특성이 있습니다.

BaseStream.unordered()메서드 를 사용하여 스트림이 정렬되지 않았는지 확인할 수 있습니다 . 예를 들면 :

```
Map<Buyer, List<Transaction>> salesByBuyer
    = txns.parallelStream()
          .unordered()
          .collect(groupingByConcurrent(Transaction::getBuyer));
```

(여기서는 Collectors.groupingByConcurrent(java.util.function.Function<? super T, ? extends K>)의 동시 해당 groupingBy)입니다.

주어진 키에 대한 요소가 소스에 나타나는 순서대로 나타나는 것이 중요하다면 순서가 동시 삽입의 피해 중 하나이기 때문에 동시 감소를 사용할 수 없습니다. 그런 다음 순차적 감소 또는 병합 기반 병렬 감소를 구현하도록 제한됩니다.

연관성

연산자 또는 함수 op는 다음이 유지되는 경우 연관 됩니다.

```
(a op b) op c == a op (b op c)
```

이것을 병렬 평가에 대한 중요성은 이것을 네 가지 용어로 확장하면 알 수 있습니다.

```
a op b op c op d == (a op b) op (c op d)
```

따라서 (a op b)와 병렬로 평가 (c op d)한 다음 op결과 를 호출 할 수 있습니다.

연관 연산의 예로는 숫자 더하기, 최소 및 최대, 문자열 연결이 있습니다.

저수준 하천 건설

지금까지 모든 스트림 예제는 Collection.stream()또는 같은 방법 Arrays.stream(Object[]) 을 사용하여 스트림을 얻었습니다. 이러한 스트림 베어링 방법은 어떻게 구현됩니까?

클래스 StreamSupport에는 스트림을 생성하기위한 여러 가지 저수준 메서드가 있으며 모두 Spliterator. spliterator는의 병렬 아날로그입니다 Iterator. 순차 전진, 대량 순회, 입력의 일부를 병렬로 처리 할 수있는 다른 분할기로 분할하는 기능을 지원하는 요소 모음 (아마도 무한대)을 설명합니다. 가장 낮은 수준에서 모든 스트림은 분할기에 의해 구동됩니다.

분할기를 구현할 때 여러 가지 구현 선택 사항이 있으며, 거의 모두 해당 분할기를 사용하는 스트림의 구현 단순성과 런타임 성능 사이의 균형입니다. 분할자를 만드는 가장 간단하지만 성능이 가장 떨어지는 방법은를 사용하여 반복자에서 분할자를 만드는 것입니다 Spliterators.spliteratorUnknownSize(java.util.Iterator, int). 이러한 분할기가 작동하는 동안 크기 조정 정보 (기본 데이터 세트의 크기)를 잃고 단순한 분할 알고리즘으로 제한되기 때문에 병렬 성능이 저하 될 수 있습니다.

고품질 분할자는 균형 잡힌 알려진 크기 분할, 정확한 크기 조정 정보 및 characteristics실행을 최적화하기 위해 구현에서 사용할 수있는 기타 여러 분할기 또는 데이터를 제공합니다.

변경 가능한 데이터 소스를위한 분할기는 추가적인 문제가 있습니다. 분할기가 생성 된 시간과 스트림 파이프 라인이 실행되는 시간 사이에 데이터가 변경 될 수 있기 때문입니다. 이상적으로 스트림에 대한 분할자는 IMMUTABLE또는 의 특성을보고합니다 CONCURRENT. 그렇지 않은 경우 늦게 구속 되어야합니다 . 소스가 권장 분할자를 직접 제공 할 수없는 경우를 사용하여 간접적으로 분할자를 제

공하고 Supplier의 Supplier-accepting 버전을 통해 스트림을 구성 할 수 stream()있습니다. 분배기는 하천 파이프 라인의 터미널 운영이 시작된 후에 만 공급자로부터 획득됩니다.

이러한 요구 사항은 스트림 소스의 변형과 스트림 파이프 라인 실행 간의 잠재적 인 간섭 범위를 크게 줄입니다. 원하는 특성을 가진 분할 자 기반 스트림 또는 공급 업체 기반 팩토리 형식을 사용하는 스트림은 터미널 작업을 시작하기 전에 데이터 소스의 수정에 영향을받지 않습니다 (스트림 작업에 대한 동작 매개 변수가 비에 대한 필수 기준을 충족하는 경우). 간섭 및 무국적). 자세한 내용은 비 간섭 을 참조하십시오.

이후:
1.8

인터페이스 요약

상호 작용	기술
BaseStream <T, S는 BaseStream <T, S >> 를 확장합니다.	순차 및 병렬 집계 작업을 지원하는 요소 시퀀스 인 스트림 용 기본 인터페이스입니다.
수집가 <T, A, R>	입력 요소를 변경 가능한 결과 컨테이너에 누적 하는 변경 가능한 축소 작업 으로, 모든 입력 요소가 처리 된 후 선택적으로 누적 된 결과를 최종 표현으로 변환합니다.
DoubleStream	순차 및 병렬 집계 작업을 지원하는 기본 이중 값 요소 시퀀스입니다.
DoubleStream.Builder	위한 가변 빌더 DoubleStream.
IntStream	순차 및 병렬 집계 작업을 지원하는 기본 int 값 요소의 시퀀스입니다.
IntStream.Builder	위한 가변 빌더 IntStream.
LongStream	순차 및 병렬 집계 작업을 지원하는 기본 long-valued 요소의 시퀀스입니다.
LongStream.Builder	위한 가변 빌더 LongStream.
스트림 <T>	순차 및 병렬 집계 작업을 지원하는 일련의 요소입니다.
Stream.Builder <T>	위한 가변 빌더 Stream.

수업 요약

수업	기술
수집가	그 구현은 Collector요소를 컬렉션으로 축적하고 다양한 기준에 따라 요소를 요약하는 등 다양한 유용한 축소 작업을 구현합니다.
StreamSupport	스트림을 만들고 조작하기위한 저수준 유틸리티 메서드.

열거 형 요약

열거 형	기술
수집가. 특성	Collector축소 구현을 최적화하는 데 사용할 수있는의 속성을 나타내는 특성입니다 .