PCD2018 - 17

PARALLEL STREAMS

OPPORTUNITÀ

Lo Stream è una diversa rappresentazione dell'iterazione su di un insieme di oggetti.

In particolare, rappresenta l'iterazione su di un insieme di cardinalità non nota, potenzialmente infinita.

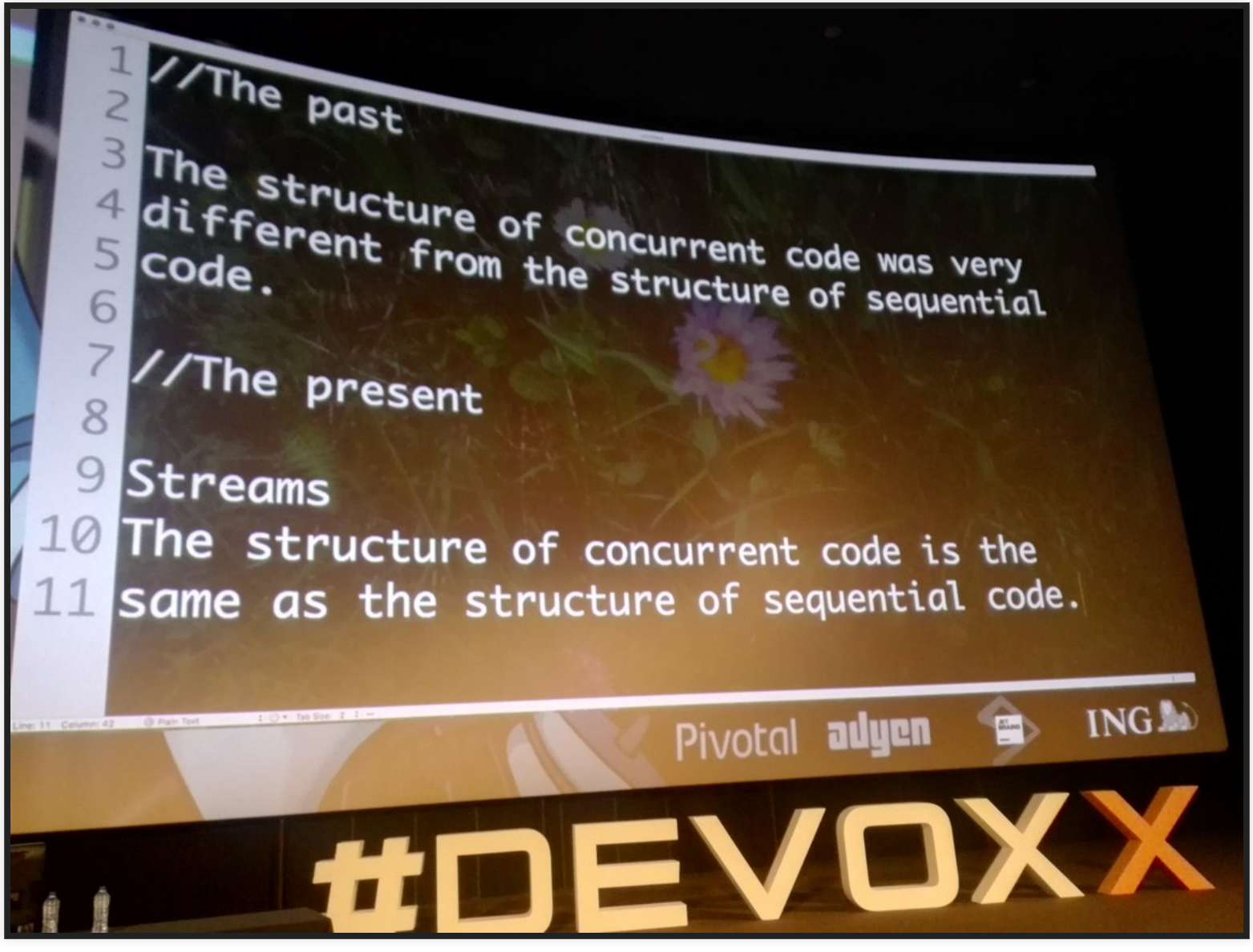
In cambio di alcune restrizioni sulle operazioni possibili e della cessione del controllo sull'ordine di iterazione otteniamo:

- una API per comporre passi di elaborazione riusabili su di una successione di oggetti

- una reificazione della trasformazione, che ne permette il riuso su sorgenti diverse ed il test in isolamento

- la promozione di un approccio funzionale al trattamento iterativo dei dati

- la disponibilità di uno stile espressivo e dichiarativo per descrivere una ampia classe di elaborazioni


1 //The past
2
3 The structure of concurrent code was very
4 different from the structure of sequential
5 code.
6
7 //The present
8
9 Streams
10 The structure of concurrent code is the
11 same as the structure of sequential code.
Pivotal
adyen
ING
#DEVOXX

- un modello di esecuzione in cui esecuzione sequenziale e parallela non richiedono modifiche al codice

In cosa sono differenti gli Stream dalle Collezioni?

La caratteristica principale di una Collezione è la performance dell'accesso ai contenuti.

L'algoritmo che usa quei contenuti è completamente estraneo a queste considerazioni.

Uno Stream richiede la definizione dell'algoritmo di calcolo che avverrà sopra i suoi elementi.

Questi ultimi non possono essere acceduti direttamente, e l'algoritmo non ha nessun controllo sul loro recupero.

Lo Stream perciò ha un certo il controllo sull'algoritmo, e può usare questo controllo per prendere decisioni su come eseguirlo.

STREAM FLAGS

La sorgente di uno Stream può dichiarare alcune *caratteristiche* che gli operatori intermedi possono verificare e che l'operazione terminale usa per prendere decisioni sulla esecuzione.

Flag	Caratteristica
CONCURRENT	Parallelizzabile
DISTINCT	Elementi distinti
IMMUTABLE	Immutabile durante il consumo

Flag	Caratteristica
NONNULL	Elementi non nulli
ORDERED	Elementi ordinati
SIZED	Dimensione nota

Flag	Caratteristica
SORTED	Ordinamento definito
SUBSIZED	Suddivisioni di dimensione nota

L'operazione terminale ha piena visibilità, prima di cominciare a lavorare, di quali sono le caratteristiche della pipeline di esecuzione, e può quindi prendere decisioni in merito.

pcd2018.streams.CountStream

```
long cnt = IntStream.range(1, 20)
  // .parallel()
  .map((i) -> {
  System.out.println(i);
  return i * 2;
}).limit(5).count();
System.out.println(">" + cnt);
```

Una pipeline può essere eseguita in modo sequenziale o parallelo semplicemente configurandola come tale.

```
/**
 * Base interface for streams, which are sequences
 * of elements supporting sequential and parallel
 * aggregate operations.
 */
interface BaseStream<T,S extends BaseStream<T,S>>
```

```
/**
 * Returns an equivalent stream that is parallel.
 * May return itself, either because the stream
 * was already parallel, or because the underlying
 * stream state was modified to be parallel.
 */
S parallel()
```

```
/**
 * Returns an equivalent stream that is sequential.
 * May return itself, either because the stream
 * was already sequential, or because the underlying
 * stream state was modified to be sequential.
 */
S sequential()
```

Una pipeline **ORDERED** garantisce di ritornare gli elementi nell'ordine in cui sono stati emessi.

I passi intermedi che mantengono questa caratteristica non modificano l'ordine.

L'ordinamento è conservato, se possibile, anche dopo una esecuzione parallela

pcd2018.streams.OrderedStream

```
IntStream.range(1, 20).parallel().map((i) -> {
  System.out.println(i + " " +
    Thread.currentThread().getName());
  return i * 2;
}).forEachOrdered((i) -> {
  System.out.println(">>> " + i);
});
```

Una pipeline **SORTED** mantiene gli elementi ordinati secondo il loro ordinamento naturale o dato da un `Comparator`.

La caratteristica **DISTINCT** garantisce che non ci siano due elementi uguali secondo `equals()`.

Una pipeline **SIZED** garantisce una dimensione nota. Se inoltre è **SUBSIZED** vuol dire che può fornire efficientemente dei sottoinsiemi dimensionati per l'esecuzione parallela.

Le operazioni che passiamo ai passi intermedi devono essere:

- non invasive: non devono modificare od interferire con gli elementi dello stream
- (nella maggior parte dei casi) prive di stato interno: devono essere cioè *funzioni pure*

Alcune operazioni sono identificate come *short-circuiting*: significa che possono interrompere l'esecuzione della pipeline prima dell'esame di tutti gli elementi.

```
/**
 * Returns whether any elements of this stream match
 * the provided predicate. May not evaluate the
 * predicate on all elements if not necessary for
 * determining the result.
 */
boolean anyMatch(Predicate<? super T> predicate)
```

```
/**
 * Returns an Optional describing some element of the
 * stream, or an empty Optional if the stream is empty.
 * The behavior of this operation is explicitly
 * nondeterministic; it is free to select any element
 * in the stream.
 */
Optional<T> findAny()
```

pcd2018.streams.CandidateNumber

```
public class CandidateNumber {
  public final int n;
  public final List<Integer> divisors;

  CandidateNumber(int n, List<Integer> divisors) {
    this.n = n;
    this.divisors = divisors;
  }
}
```

pcd2018.streams.Divisors

```
public class Divisors implements
  Function<Integer, CandidateNumber> {

  @Override
  public CandidateNumber apply(Integer n) {

    ...

    return new CandidateNumber(n, divs);
  }
}
```

pcd2018.streams.Perfect

```
public class Perfect implements Predicate<CandidateNumber> {

  @Override
  public boolean test(CandidateNumber c) {
    Integer sum = c.divisors.stream()
        .collect(Collectors.summingInt((Integer n) -> n));
    return (sum + 1) == c.n;
  }

}
```

pcd2018.streams.PerfectStream

```
List<CandidateNumber> match = IntStream.range(30, 100000)
  .boxed().parallel().map(new Divisors())
  .filter(new Perfect()).findAny().stream()
  .collect(Collectors.toList());

match.forEach(x -> { System.out.println(x); });
```

Altre operazioni sono dette *stateful*: può necessitare di consumare tutto o gran parte dell'input per poter emettere l'output mantenendo le caratteristiche desiderate (per es. ordinamento).

```
/**
 * Returns a stream consisting of the elements of this
 * stream, truncated to be no longer than maxSize in
 * length.
 */
Stream<T> limit(long maxSize)
```

pcd2018.streams.AllPerfectStream

```
IntStream.range(10, 10000).boxed().parallel()
  .map(new Divisors()).filter(new Perfect()).limit(2)
  .forEach((CandidateNumber c) -> {
    System.out.println(">>> " + c.toString());
  });
```

SPLITITERATOR

Per esprimere una sorgente in grado di essere parallelizzabile non è sufficiente l'API esposta da `Supplier` o `Iterator`.

Servono infatti dei metodi che consentano alla sorgente di esplicitare le opportunità di suddivisione dello stream in rami di esecuzione indipendenti.

```
/**
 * An object for traversing and partitioning elements
 * of a source.
 *
 */
public interface Spliterator<T>
```

Il metodo di avanzamento diventa `tryAdvance()`,
che ribalta il funzionamento dell'iterator:
non è l'utilizzatore che ottiene il nuovo elemento, ma è lo stream che fornisce l'elemento al codice che deve operarci sopra.

```java
/**
 * If a remaining element exists, performs the given
 * action on it, returning true; else returns false.
 *
 */
boolean tryAdvance(Consumer<? super T> action)
```

`Splititerator` aumenta l'espressività di `Iterator` aggiungendo metodi per:

- stimare gli elementi rimanenti
- esplicitare le caratteristiche della sorgente
- attraversare in massa gli elementi rimanenti
- suddividere l'iterazione in più rami

```
/**
 * Returns an estimate of the number of elements that
 * would be encountered by a forEachRemaining()
 * traversal, or returns Long.MAX_VALUE if infinite,
 * unknown, or too expensive to compute.
 *
 * @return the estimated size
 */
long estimateSize()
```

```
/**
 * Returns a set of characteristics of this
 * Spliterator and its elements.
 *
 * @return a representation of characteristics
 */
int characteristics()
```

```
/**
 * Performs the given action for each remaining element,
 * sequentially in the current thread, until all elements
 * have been processed or the action throws an exception.
 *
 */
default void forEachRemaining(Consumer<? super T> action)
```

```
/**
 * If this spliterator can be partitioned, returns a
 * Spliterator covering elements, that will, upon return
 * from this method, not be covered by this Spliterator.
 *
 * @return a Spliterator covering some portion of the
 *           elements, or null if this spliterator cannot
 *           be split
 *
 */
Spliterator<T> trySplit()
```

Con il supporto di queste funzioni, lo stream può pianificare efficientemente l'esecuzione delle operazioni terminali.

```
/**
 * Performs a reduction on the elements of this stream,
 * using the provided identity value and an associative
 * accumulation function, and returns the reduced value.
 *
 * @param identity the identity value for the accumulating
 *                 function
 * @param accumulator an associative, non-interfering,
 *                    stateless function for combining
 *                    two values
 * @result the result of the reduction
 *
 */
T reduce(T identity, BinaryOperator<T> accumulator)
```

pcd2018.streams.StreamReduce

```
int res = IntStream.range(1, 1001).parallel().
  reduce(0, (a, b) -> {
    System.out.println(a + "+" + b + "=" + (a + b)
      + " " + Thread.currentThread().getName());
    return a + b;
  });
System.out.println(">>>> " + res);
```

```
/**
 * Performs a mutable reduction operation on the elements
 * of this stream using a Collector.
 *
 * @R the type of the result
 * @A the intermediate accumulation type of the Collector
 * @param the Collector describing the reduction
 * @result the result of the reduction
 */
<R,A> R collect(Collector<? super T,A,R> collector)
```

pcd2018.streams.StreamCollector

```
int res = IntStream.range(1, 1001).boxed().parallel()
  .collect(Collectors.summingInt((i) -> i));
System.out.println(">>>> " + res);
```

COLLECTOR

Nell'operazione di riduzione gestita da `reduce` l'accumulazione del risultato avviene creando nuovi valori. Questo però in certi casi non è efficiente.

pcd2018.streams.StringReduce

```
String res = IntStream.range(1, 1001).boxed()
  .map((i) -> i.toString())
  .parallel().reduce("", String::concat);
System.out.println(">>>> " + res);
```

L'interfaccia `Collector` permette di gestire una riduzione dove l'accumulatore è un oggetto mutabile per ragioni di efficienza.

```
/**
 * A mutable reduction operation that accumulates input
 * elements into a mutable result container.
 *
 * @T the type of input elements to the reduction
 *       operation
 * @A the mutable accumulation type of the
 *       reduction operation
 * @R the result type of the reduction operation
 */
interface Collector<T,A,R>
```

La classe `Collectors` permette di produrre dei `Collector` a partire dagli elementi di base:

- un `Supplier<A>` del contenitore di risultato
- un `BiConsumer<A,T>` che accumula un elemento nel contenitore

- un `BinaryOperator<A>` che combina due contenitori parziali
- un `Function<A,R>` che dal contenitore ottiene il risultato finale

Combinando questi elementi lo `Stream` ha tutte le parti della strategia per applicare l'algoritmo, e ha le caratteristiche della sorgente per calcolare come organizzare l'esecuzione.

pcd2018.streams.StringCollector

```
String res = IntStream.range(1, 1001).boxed()
  .map((i) -> i.toString()).parallel().collect(
    // supplier
    () -> new StringBuffer(),
    // accumulator
    (acc, el) -> acc.append(el),
    // combiner
    (resA, resB) -> resA.append(resB))
  .toString();
System.out.println(">>>> " + res);
```

PARALLEL STREAMS

Gli Stream sono quindi un'ottima astrazione per modellare in modo semplice algoritmi su collezioni di elementi.

Il nostro codice può essere eseguito in parallelo con un semplice cambio di configurazione.

Lo Stream tuttavia di default decide autonomamente quanto parallelismo usare, attraverso il `ForkJoinPool`.

Quanto parallelismo possiamo richiedere?

Blocking factor:
Intensità del calcolo di un algoritmo

Un algoritmo con `BF=0`
occupa costantemente la CPU.

Un algoritmo con `BF=1`
è costantemente in attesa di I/O.

$$\#threads <= \frac{\#of\ cores}{1-BF}$$

Se il nostro algoritmo effettua molta I/O può essere utile modificare l'impostazione standard del `ForkJoinPool` per aumentare il numero di thread disponibili.

Dr. Venkat Subramaniam: The Power and Perils of Parallel Streams

https://www.youtube.com/watch?v=0-f-1Cx0HaU

Allo stesso tempo l'uso dello Stream come modello di calcolo può grandemente semplificare l'aspetto e quindi la leggibilità e la manutenibilità del nostro codice:

```java
public static double compute1(int n, int k) {
  int index = n;
  int count = 0;
  double result = 0;

  while(count < k) {
    if(isPrime(index)) {
      result += Math.sqrt(index);
      count++;
    }
    index++;
  }
  return result;
}
```

```
public static double compute2(int n, int k) {
  return Stream.iterate(n, e -> e + 1)
               .filter(Sample::isPrime)
               .limit(k)
               .mapToDouble(Math::sqrt)
               .sum();
}
```

APPROFONDIMENTI

Parallel and Asynchronous Programming with Streams and CompletableFuture by Venkat Subramaniam (Devoxx 2017)

https://www.youtube.com/watch?v=IwJ-SCfXoAU