Java 8 Stream behind the scenes: internal iteration, lazines and fluent API

Modelação e Padrões de Desenho

2016

`quiny.Queryable<T>`

quiny provides a **tiny** implementation of the class `Queryable<T>`, equivalent to java 8 `Stream<T>`. `Queryable<T>` is a **concise** and **functional** implementation of an equivalent API to the `Stream<T>`, which preserves the **internal iteration** approach, the **laziness** behavior and the **fluent idiom**. This solution answers the question: *How can I implement a lazy iterator in Java 8?*
To achieve a short implementation I suppressed from `Stream<T>` the partitioning feature (meaning that `Queryable<T>` does NOT support parallel processing). So, I took off all the infrastructure panoply and stayed only with the essential backbone that provides a query API.
Figure 1, shows an example using `Queryable<T>` that is equivalent to the use of `Stream<T>`. You can replace the `Queryable.of(dataSrc)` call with `dataSrc.stream()` and you will get the same result. You can try it your self by copying the implementation code of section Queryable at a glance and execute the example of Figure 1. Challenge yourself and add new query methods to `Queryable<T>`. All you have to do is just return a new lambda that implements the `tryAdvance(action)` logic.

Figure 1:

```
Collection<String> dataSrc = ...  // something
Queryable.of(dataSrc)             // <=> dataSrc.stream()
   .filter(w -> !w.startsWith("-"))
   .distinct()
   .map(String::length)
   .limit(5)
   .forEach(System.out::println)
```

Queryable at a glance

For now I will just show you how you can develop a very short implementation of `map()`, `limit()` and `forEach()`. This implementation is **purely functional** and does not reuse any code of the new default methods provided in Java 8. Moreover it preserves the internal iteration approach; it is lazy and also provides a fluent idiom. You can copy paste it and test it, just like it is.

```
@FunctionalInterface
interface Queryable<T>{

  abstract boolean tryAdvance(Consumer<? super T> action); // <=> Spliterator::tryAdvance

  static <T> boolean truth(Consumer<T> c, T item){
    c.accept(item);
    return true;
  }

  public static <T> Queryable<T> of(Iterable<T> data) {
    final Iterator<T> dataSrc = data.iterator();
    return action -> dataSrc.hasNext() && truth(action, dataSrc.next());
  }
```

```
  public default void forEach(Consumer<? super T> action) {
    while (tryAdvance(action)) { }
  }

  public default <R> Queryable<R> map(Function<T, R> mapper) {
    return action -> tryAdvance(item -> action.accept(mapper.apply(item)));
  }

  public default Queryable<T> limit(long maxSize) {
    final int[] count = {0};
    return action -> count[0]++ < maxSize ? tryAdvance(action) : false;
  }
}
```
In the next sections I give a deeper and more detailed explanation about this solution.

Introduction

First, just to give you a brief about the implementation complexity of `Stream<T>`: if you look into `stream()` default method you will find the inner `ReferencePipeline.Head<E_IN, E_OUT>` implementation of `Stream<T>` that in turns depends of a `Spliterator<?>` source stored in a `sourceSpliterator` field inherited from `AbstractPipeline`. For collections the `Spliterator<?>` is an instance of `IteratorSpliterator`. On the other hand, each intermediate operation (such as, `filter`, `map`, etc) returns a new instance of `StatelessOp<E_IN, E_OUT>` which in turn creates an new instance of `Sink.ChainedReference<…>` that is used to conduct values through the stages of a stream pipeline, with additional methods to manage size information, control flow, etc. Moreover you still have to dig into the additional infrastructure that supports partitioning for parallel processing.
So, I would like to take off all this infrastructure panoply and stay only with the essential backbone that allows the **internal iteration** feature, **laziness** behavior and **fluent API**.

To that end I made my own implementation of `Queryable<T>` with a similar API and preserving these three characteristics. This `Queryable<T>` can be used in the same manner of `Stream<T>` according to the example of Figure 1, or inter-exchanging the order between the intermediate operations: `distinct()`, `filter()`, `map()` and `limit()`.
If you are not interested in the path that founded `Queryable<T>` solution, then you can jump directly to the implementation of the last version of Queryable
In this repository I provide three implementations of `Queryable<T>` in three different braches:

- version1--oo – following an object oriented approach.
- version2 – version 1 simplification
- version3-functional – a functional approach without instance fields, no objects instantiation an no constructors.

Scope

In the following sections I will explain each step that conduct my own implementation of **Queryable**. The `Queryable<T>` provides the same features of Figure 1 and preserves the same characteristics of **internal iteration**, **laziness** and **fluent API**. Of course it is not my goal to achieve a better solution than the existing Java 8 implementation. On the other hand, my implementation guidelines were:

1. **simplicity**
2. **readability**
3. **functional style** (provided that it preserves 1 and 2 (otherwise avoid functional))
4. **avoid auxiliary classes**

5. **no instance fields**
6. **no mutable shared state**.

Until this date I did not find any other implementation that fulfills these requisites. **REMEMBER**: It is not my intention to replace the `Stream<T>`.
Last note: I will not deal with the parallel computing feature provided by `Stream<T>`. That is out of the scope of this post and you will find a lot of information about that topic. Nevertheless, if you are considering to use `Stream<T>` for task parallelization you should read the excellent article [A Java™ Parallel Calamity]().

`Spliterator<T>` as the basis for `Stream<T>`

This section answers the following questions:

- Why `Spliterator<T>` is the basis for `Stream<T>`?
- Why `Spliterator<T>` instead of `Iterator<t>`?

Trying to understand the laziness characteristic of Java 8 `Stream<T>`, I tried to figure out an implementation that supports and enables the `Stream`'s lazy behavior. You can do it with through the interfaces `Iterable<T>` and `Iterator<T>` just like [Guava]() does for its class `Iterables`, which provides the same query methods (such as `map`, `filter`, etc) as the `Stream<T>`. Or even, the seminal [Linq .net framework]() based on the interfaces `IEnumerable<T>` and `IEnumerator<T>` (equivalent to the *iterator* interfaces of Java).
But if you start digging on `stream()` default implementation you will easily fall in the inevitable `Spliterator<T>`. You will find that `Spliterator<T>` is the core essential part of the `Stream<T>` implementation. On a simple point of view you can see a `Stream<T>` as a combination of **default methods** with **`Spliterator<T>`**.

```
Stream<T> ~ _default methods_  + Spliterator<T>
```

Default methods provide the query utilities (such as `filter`, `map`, etc) in a fluent style idiom and the `Spliterator<T>` **provides the ability of traversing elements from a data source**.
And this fact may raise your first question: **Why a new kind of iterator--`Spliterator<T>`???** Why not the already existent `Iterator<T>` and its abstract factory `Iterable<T>`? Easily you will find a lot of posts that digress around the evident answer provided by the Java API docs:
A Spliterator may also partition off some of its elements (using trySplit()) as another Spliterator, to be used in **possibly-parallel operations**.
And this is the key insight that enables one of so promoted features of `Stream<T>`: **support to parallel aggregate operations**. (which is not its main advantage IMHO).
But `Spliterator` also provides a different iteration approach called **internal iteration**, where *you specify what to do* on each iteration (through a function argument of type `Consumer<T>`), instead of *doing something with the items that you got* from each iteration. So, you will *NOT get the items from the spliterator* (e.g. `T item = iter.next(); compute(item)`) , you will instead *specify to the spliterator what to do with those items* (e.g. `iter.tryAdvance(item -> compute(item))`). Note in the latter case you pass another function (`Consumer<T>`) to the iteration method (`tryAdvance`). Of course, in both examples you could simplify the iteration with the Java *foreach* statement and using the `forEach` default method of `Stream<T>` (or even the `forRemaining` default method already provided

on Spliterator). But for now I just want to revisit the low level code that supports the external and internal iteration approaches.
However, you do not need an inner iterator implementation following the *internal iteration* approach to implement lazy query methods, such those ones provided by Stream<T> (e.g. map, filter, etc). You can implement those same methods using the existent Iterator<T>. For instance the query methods provided by class Iterables of Guava project are **lazy** and built on top of the interface Iterable<T> and Iterator<T>.
So, what else offers Spliterator<T>???
Note that the internal iteration API of Spliterator<T> is based on **one only method**--boolean tryAdvance(Consumer<T> action)-- instead of two methods--boolean hasNext() and T next(). This unpromoted aspect (1 only method, instead of 2) will make a big difference when we rewrite the Queryable<T> to its simplest version, because we may provide the implementation of its abstract method through a lambda expression. This simple feature will avoid the need of additional auxiliary classes.
In the next three sections we will present three versions of the implementation of the Queryable<T> proposal, from a more Object Oriented approach to a completely Functional approach without instance fields, no objects instantiation an no constructors.

Queryable -- version 1 – Object Oriented

The only visible method of Queryable on Figure 1 is the static method of() which creates "something" providing the query methods: filter, distinct, etc... If we add these query methods as instance methods of the same class Queryable<T> then we may define the method of() with a return type of Queryable<T>. And in turn to provide a fluent API the query methods may also return an instance of Queryable<T>. So in Figure 3 we depict the skeleton that is the basis for the implementation of Queryable<T>.

Figure 3:

```
public class Queryable<T> {

    public static <T> Queryable<T> of(Collection<T> data) {
        throw new NotImplementedException();
    }

    public Queryable<T> distinct() {
        throw new NotImplementedException();
    }

    public Queryable<T> filter(Predicate<T> p) {
        throw new NotImplementedException();
    }

    public <R> Queryable<R> map(Function<T, R> mapper) {
        throw new NotImplementedException();
    }

    public Queryable<T> limit(long maxSize) {
        throw new NotImplementedException();
    }

    public void forEach(Consumer<T> action) {
        throw new NotImplementedException();
    }
}
```

Despite the `forEach` method, which is a **terminal operation**, the remaining query methods are **intermediate operations** which are defined as operations that return a new `Queryable`. This idea follows the same principle of the intermediate operations of `Stream<T>` that are described in the [Java API stream package summary]() as:

Intermediate operations return a new stream. They are always *lazy*; executing an intermediate operation such as `filter()` does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

Each `Queryable<T>` object returned by a query operation conveys elements from a source such as a data structure (i.e. the `data` variable of [Figure 1]()) or from the `Queryable<T>` object returned by the previous operation. For example, the source for the `Queryable<T>` object returned by `distinct()` of [Figure 1]() is the previous `Queryable<T>` object returned by `filter()`.

So we need some way to traverse the elements of each `Queryable<T>` object independently of its kind of source. Note that the source could be in memory or generated on demand or even the result of the previous computation. So, following the Java 8 guidelines we will use a `Spliterator<T>` as the data source for each `Queryable<T>`.

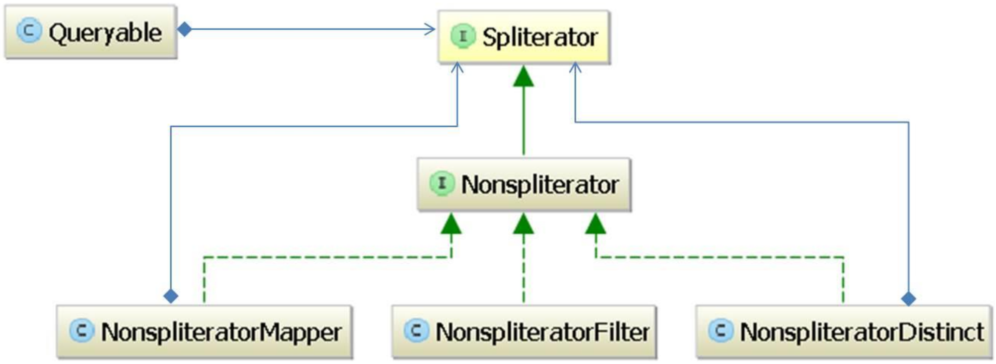
Now on [Figure 4]() we add a `dataSrc` field of type `Spliterator<T>` and, in turn, each method returns a new `Queryable<T>` object instantiated with a new instance of `Spliterator<T>` (I made all implementations of `Spliterator<T>` as subclasses of `Nonspliterator<T>` (explained later)). Each implementation of `Spliterator<T>` applies a specific query policy to the items returned by the previous `Spliterator<T>` (Note that each instance of `Spliterator<T>` receives the current `dataSrc` as the constructor argument). For each query method we will have a different implementation of `Spliterator<T>` with different requirements. For instance the `NonspliteratorFilter` returned by `filter` requires a `Predicate` whereas the `NonspliteratorMapper` returned by `map` requires a `Function<T,R>`.

Figure 4:

```
public class Queryable<T> {

    private final Spliterator<T> dataSrc;

    public Queryable(Spliterator<T> dataSrc) {
        this.dataSrc = dataSrc;
    }

    public static <T> Queryable<T> of(Collection<T> data) {
        return new Queryable<T>(new NonspliteratorIterator(data.iterator()));
    }

    public Queryable<T> distinct() {
        return new Queryable<>(new NonspliteratorDistinct<>(dataSrc));
    }

    public Queryable<T> filter(Predicate<T> p) {
        return new Queryable<>(new NonspliteratorFilter<>(dataSrc, p));
    }

    public <R> Queryable<R> map(Function<T, R> mapper) {
        return new Queryable<>(new NonspliteratorMapper<>(dataSrc, mapper));
    }

    public Queryable<T> limit(long maxSize) {
        return new Queryable<>(new NonspliteratorLimited<>(dataSrc, maxSize));
```

```
    }

    public void forEach(Consumer<T> action) {
        while (dataSrc.tryAdvance(action)) ;
    }
}
```

`Nonspliterator` is the base type for all auxiliary subclasses of `Spliterator<T>`. I call it *Non* spliterator because it does not allow its division in sub iterators. It provides a default implementation for the partitioning methods (`trySplit()`, `estimateSize()` and `characteristics()`) that avoids partitioning. Remember that parallel processing is out of the scope.



Maybe we could refactor code and include a `srcIterator` field in `Nonspliterator<T>`, but in the case of `NonspliteratorMapper` the `tryAdvance()` method receives a `Consumer<R>` that is parametrized with a type `R` different of `T`. So a generic definition of `Nonspliterator<T>` with a `srcIterator` field will required two type arguments instead of one, which will increase complexity and reduce readability. So, for simplicity (remember the guidelines) I preferred to repeat the `srcIterator` field in all subclasses of `Nonspliterator<T>` (moreover, this is not the final implementation).

Finally, **don't be tempted to return this on query methods**. You may be thinking on turning `dataSrc` into a mutable field and thus avoid the instantiation of a new `Queryable<T>` object on each query method. But, that is an error prone approach. I do not want to extend this post substantiating the reasons. I will just give you another citation of the [Java API stream package summary] 2-stream-summary as:

Functional in nature. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection

Queryable -- version 2 – abstracts `tryAdvance()`

version2 – version 1 simplification

Queryable -- version 3 – Functional

version3-functional

- How to combine default methods and lambdas to implement a `Spliterator<T>`?
- How to simply provide laziness and fluent query methods on top of a `Spliterator<T>`?