Java Streams, Part 3: Streams の内幕

java.util.stream の内部の仕組みを理解する

Brian Goetz
Java Language Architect
Oracle

2016年 10月 27日

このシリーズでは、Java 言語アーキテクトである Brian Goetz が、Java SE 8 で導入された Java Streams ライブラリーの詳細を解説します。この `java.util.stream` パッケージは、ラムダ式の力を利用することによって、コレクションや配列などのさまざまなデータ・セットに対し、関数型スタイルのクエリーを簡単に実行できるようにします。今回の記事で、クエリーを微調整して可能な限り効率化するにはどのようにするのかを学んでください。

このシリーズの他の記事を見る

このシリーズの最初の 2 回の記事では、Java SE 8 で導入された `java.util.stream` ライブラリーを利用して、データ・セットに対するクエリーを宣言型によって簡単に表現するにはどのようにするのかを説明しました。多くの場合、このライブラリーはどうすればクエリーを効率的に実行できるのかを、ユーザーによる手助けなしに突き止めます。けれどもパフォーマンスが極めて重要な場合には、パフォーマンスを低下させる可能性がある原因を排除できるよう、このライブラリーが内部でどのように機能するのかを理解しておくことが重要となります。第 3 回となるこの記事では、Streams の実装がどのように機能するのかを説明し、宣言型の手法によって可能になる最適化をいくつか紹介します。

ストリーム・パイプライン

ストリーム・パイプラインは、ストリーム・ソース、ゼロ以上の中間処理、そして単一の終端処理からなります。ストリーム・ソースとしては、コレクション、配列、ジェネレーター関数を使用できます。さらに、内在する要素に適切にアクセスできるようになっているデータ・ソースであれば、どのデータ・ソースでもストリーム・ソースとして使用できます。中間処理は、ストリームを他のストリームに変換します。その方法としては、要素をフィルタリングする (`filter()`)、要素の形を変換する (`map()`)、要素をソートする (`sorted()`)、ストリームを特定のサイズにする (`limit()`) などが挙げられます。終端処理には、集約 (`reduce()`、`collect()`)、検索 (`findFirst()`)、繰り返し処理 (`forEach()`) などがあります。

このシリーズについて
`java.util.stream` パッケージを使用すると、コレクションや配列などのさまざまなデータ・ソースに対する一括処理を、おそらく並列でも実行できる、簡潔な宣言型の処理として表現することができます。このシリーズでは、Java 言語アーキテクトである Brian Goetz

が、Streams ライブラリーについて包括的に解説し、このライブラリーを最大限活用するにはどのようにするのかを説明します。

ストリーム・パイプラインは、遅延処理として構成されます。ストリーム・ソースを構成するということは、ストリームの要素を計算することではなく、必要に応じて要素を見つける方法を取り込むということです。同様に、中間処理を呼び出すということは、要素に対する計算を行うことではなく、ストリームの記述の最後に別の処理を追加することでしかありません。終端処理が呼び出された時点で、パイプラインはその実際の処理として要素を計算し、中間処理、そして終端処置を適用します。この実行手法は、いくつかの興味深い最適化を可能にします。

ストリーム・ソース

ストリーム・ソースを記述するには、`Spliterator` という抽象化を使用します。その名前が示すように、`Spliterator` はソースの要素にアクセスするという動作 (繰り返し処理) と、できる場合は入力ソースを並列実行用に分解するという動作 (分割) を結合したものです。

`Spliterator` には `Iterator` と同じ動作が含まれるものの、Spliterator は `Iterator` を継承するのではなく、別の手法を使用して要素にアクセスします。`Iterator` には `hasNext()` および `next()` という 2 つのメソッドがあり、次の要素にアクセスすると、この両方のメソッドが (必要ではないのに) 呼び出されることがあります。そのため、`Iterator` を正しくコーディングするには、ある程度の防御的かつ重複するコーディングが必要になります (クライアントが `hasNext()` を呼び出さずに `next()` を呼び出した場合や、`hasNext()` を 2 回呼び出した場合を考えてみてください)。さらに、2 つのメソッドを持つプロトコルには一般にかなりのステートフルネスが必要であり、1 つ前の要素を覗いたり (そして前もって覗いたかどうか経過を追ったり) するなどしなければなりません。これらの要件が重なって、要素ごとのアクセスには、かなりのオーバーヘッドが生じることになります。

Java 言語内でラムダ式を使用すると、一般により効率的な手法を `Spliterator` での要素アクセスに採用できるようになり、正しいコードを作成するのも容易になります。`Spliterator` には、要素にアクセスするためのメソッドが 2 つあります。

```
boolean tryAdvance(Consumer<? super T> action);
void forEachRemaining(Consumer<? super T> action);
```

`tryAdvance()` メソッドは、単一の要素を処理しようとします。要素が残っていなければ、`tryAdvance()` は `false` を返すだけです。要素がある限り、このメソッドはカーソルを進め、現在の要素を指定のハンドラーに渡して、`true` を返します。`forEachRemaining()` メソッドは、要素を 1 つずつ指定のハンドラーに渡すことで、残っている要素のすべてを処理します。

並列分解の可能性について説明するまでもなく、`Spliterator` 抽象化が「より有効なイテレーター」であることは明らかです。Iterator より、作成するのも使用するのも簡単で、通常は要

素アクセスごとのオーバーヘッドも小さくなるためです。ただし、`Spliterator` 抽象化は並列分解にも及びます。`Spliterator` は残りの要素のシーケンスを記述し、`tryAdvance()` または `forEachRemaining()` 要素アクセス・メソッドを呼び出して、そのシーケンスに従って処理を進めます。2 つのスレッドがそれぞれ異なる入力のセクションを独立して処理できるよう、`Spliterator` にはソースを分割するための `trySplit()` メソッドが用意されています。

```
Spliterator<T> trySplit();
```

`trySplit()` の動作は、残りの要素をできるだけ同じサイズの 2 つのセクションに分割するよう試みるというものです。`Spliterator` を分割できる場合、`trySplit()` は記述された要素の最初の部分を切り取って新しい `Spliterator` を作成し (これが結果として返されます)、現行の `Spliterator` の状態を調整して、切り取られた部分に続く残りの要素の状態を記述します。ソースを分割できない場合、`trySplit()` は分割可能でないこと、そして呼び出し側が順次処理しなければならないことを表す `null` を返します。要素の出現順に意味があるソースの場合 (配列、`List`、`SortedSet` など) については、`trySplit()` がその順序を維持する必要があります。したがって、残りの要素の最初の部分を切り取って新しい `Spliterator` を作成した後、現行の Spliterator が元の順序と一致する順で要素を記述するようにしなければなりません。

これまで、JDK 内での `Collection` 実装にはすべて、高品質の `Spliterator` 実装が装備されてきました。一部のソースは、他のソースより優れた実装につながります。例えば、複数の要素を持つ `ArrayList` は、例外なく均等に分割できます。一方、`LinkedList` は常に分割しにくいソースです。ハッシュ・ベースおよびツリー・ベースのデータ・セットは、通常はある程度上手く分割できます。

ストリーム・パイプラインの構築

ストリーム・パイプラインを構築するには、ストリーム・ソースとその中間処理との間のリンクを設定したリスト表現を構成します。内部表現内では、パイプラインの各ステージをストリーム・フラグのビットマップによって記述します。ストリーム・フラグとは、ストリーム・パイプラインのそのステージで要素について既知になっている情報を記述するものです。Streams ではこれらのフラグを使用して、ストリームの構成および実行の両方を最適化します。表 1 に、ストリーム・フラグとその意味を記載します。

表 1. ストリーム・フラグ

ストリーム・フラグ	意味
`SIZED`	ストリームのサイズは既知です。
`DISTINCT`	オブジェクト・ストリームの場合は `Object.equals()`、プリミティブ型ストリームの場合は `==` を基準として、ストリームの要素は明確に区別されます。
`SORTED`	ストリームの要素は自然順を使用してソートされています。
`ORDERED`	ストリームの出現順には意味があります (「出現順」セクションを参照)。

ソース・ステージのストリーム・フラグは、Spliterator の `characteristics` ビットマップから導かれます (Spliterator はストリームより多くのフラグをサポートしています)。高品質の Spliterator 実装は、効率的に要素にアクセスして分割できるだけでなく、要素の特性も記述しています (`Set`

の要素が個々に区別されることは既知であるため、例えば HashSet の Spliterator は DISTINCT 特性をレポートします)。

> “ 場合によっては、Streams はソースおよび先行の処理に関する知識に基づいて、特定の処理を完全に省略します。 ”

それぞれの中間処理は、ストリーム・フラグに既知の影響を与えます。具体的には、中間処理によって、各フラグの設定値が設定、クリア、または維持される場合があります。例えば、filter() 処理は SORTED および DISTINCT フラグを維持する一方、SIZED フラグをクリアします。map() 処理は SORTED および DISTINCT フラグをクリアする一方、SIZED フラグを維持します。sorted() 処理は、SIZED および DISTINCT フラグを維持して、SORTED フラグを注入します。各ステージのリンク付きリスト表現を構成する際に、前のステージのフラグが現行ステージの動作に結合されて、現行ステージの新しいフラグ一式になります。

場合によっては、これらのフラグの存在によって、リスト 1 に示すストリーム・パイプライン同様に、特定の処理を完全に省略することも可能です。

リスト 1. 処理を自動的に省略できるストリーム・パイプライン

```
TreeSet<String> ts = ...
String[] sortedAWords = ts.stream()
                          .filter(s -> s.startsWith("a"))
                          .sorted()
                          .toArray();
```

ソースは TreeSet であるため、ソース・ステージのストリーム・フラグには SORTED が含まれています。filter() メソッドは SORTED フラグを維持するので、フィルタリング・ステージのストリーム・フラグにも SORTED フラグが含まれます。通常は、sorted() メソッドの結果、新しいパイプライン・ステージが構成されてパイプラインの末尾に追加され、その新しいステージが返されます。けれども要素が常に自然順でソートされることは既知であるため、sorted() メソッドは no-op となります。ソートは重複する処理になることから、このメソッドは単に前のステージ (フィルタリング・ステージ) を返すだけです (同様に、要素が DISTINCT であることが既知の場合は、distinct() 処理を完全に省略できます)。

ストリーム・パイプラインの実行

終端処理が開始された時点で、ストリーム実装は実行計画を取得します。中間処理はステートレス処理 (filter()、map()、flatMap()) とステートフル処理 (sorted()、limit()、distinct()) に分けられます。ステートレス処理とは、他の要素に関する知識がなくても要素に対して実行できる処理のことです。例えば、フィルタリング処理では現行の要素だけを調べて、その要素を含めるか除外するかを判別できます。一方、ソート処理ではすべての要素を調べてからでないと、最初に送出する要素を判断できません。

パイプラインを順次実行する場合、または並列実行するものの、パイプラインがステートレス処理だけで構成されている場合は、単一のパス内で計算することができます。それ以外の場合、パイプラインは (ステートフル処理の境界で) セクションに分割されて、複数のパス内で計算されます。

終端処理は、短絡処理 (`allMatch()`、`findFirst()`) または非短絡処理 (`reduce()`、`collect()`、`forEach()`) のいずれかです。終端処理が非短絡処理であれば、データを一括処理できます (ソース Spliterator の `forEachRemaining()` メソッドを使用して一括処理すると、各要素へのアクセスに伴うオーバーヘッドをさらに削減できます)。終端処理が短絡処理の場合は、(`tryAdvance()` を使用して) 一度に 1 つの要素を処理する必要があります。

順次実行の場合、Streams は「マシン」を構成します。マシンとは、`Consumer` オブジェクトをつなげたチェーンのことで、その構造はパイプラインの構造と一致します。これらの `Consumer` オブジェクトのそれぞれが、次のステージについての情報を持ち、要素を受信すると (または残りの要素がないことを通知されると)、ゼロ個以上の要素をチェーンの次のステージに送信します。例えば、`filter()` ステージに関連付けられている `Consumer` はフィルター述部を入力要素に適用した後、その要素を次のステージに送信する場合もあれば送信しない場合もあります。`map()` ステージに関連付けられている `Consumer` はマッピング関数を入力要素に適用した後、その結果を次のステージに送信します。`sorted()` などのステートフル処理に関連付けられている `Consumer` は、要素をバッファーに入れ、入力の最後に到達した時点で、ソートしたデータを次のステージに送信します。マシンの最終ステージでは、終端処理を実装します。終端処理が結果を生成する処理 (`reduce()` や `toArray()` など) である場合、このステージが結果のアキュムレーターの役割を果たします。

図 1 に、以下に記載するストリームと「ストリーム・マシン」のアニメーション (特定のブラウザー内では、スナップショット) を示します (図 1 の黄色、緑色、青色のブロックは、上から順番にマシンの最初のステージに入ります。各ブロックは、最初のステージ内で小さなブロックに圧縮されてから、2 番目のステージに送り込まれます。2 番目のステージでは、パックマンのようなキャラクターが黄色のブロックのそれぞれを飲み込んで、緑色と青色のブロックだけを 3 番目のステージに進めます。3 番目のステージで、コンピューター画面に圧縮された青色と緑色のブロックが交互に表示されます)。

```
blocks.stream()
      .map(block -> block.squash())
      .filter(block -> block.getColor() != YELLOW)
      .forEach(block -> block.display());
```

図 1. ストリーム・マシン (Tagir Valeev 氏の許可を得て掲載)

並列実行は上記と同じような処理を行いますが、異なる点として、単一のマシンを作成するのではなく、各ワーカー・スレッドがマシンの独自のコピーを取得します。そのコピーに、そのワーカー・スレッドが担当するデータのセクションがフィードされた後、スレッド・マシンごとの結果が他のマシンの結果にマージされて、最終結果が生成されます。

ストリーム・パイプラインの実行も、ストリーム・フラグを使うことによって最適化することができます。例えば、`SIZED` フラグは最終結果のサイズが既知であることを示します。`toArray()` 終端処理ではこのフラグを利用して、正しいサイズの配列を事前に割り当てることができます。`SIZED` フラグが存在しなければ、通常は、配列のサイズを推測しなければならず、推測が誤っている場合に備えて、可能であればデータをコピーします。

> “ パフォーマンスが非常に重要となる場合は、このライブラリーの内部の仕組みに関する知識が役立ちます。 ”

事前サイズ設定による最適化は、ストリームの並列実行内でなおさら効果を発揮します。`SIZED` フラグの他、Spliterator の `SUBSIZED` 特性は、サイズが既知であることだけでなく、Spliterator が分割される場合はそのサイズも既知であることを意味します (このことは、配列と `ArrayList` には当てはまりますが、他の分割可能なソース (ツリーなど) については必ずしも当てはまりません)。`SUBSIZED` 特性が存在する場合、並列実行内では `toArray()` 処理で単一の正しいサイズの配列を結果全体に割り当て、(それぞれに入力の異なるセクションを処理する) 個々のスレッドがその処理結果を配列の正しいセクションに直接書き込むことができます。したがって、同期する必要もコピーする必要もなくなります (`SUBSIZED` フラグがない場合、各セクションが中間配列に収集されてから最終的な場所にコピーされます)。

出現順

出現順も、このライブラリーによって最適化できるかどうかに影響する潜在的な事項として考慮しなければなりません。出現順とは、ソースが要素を分配する順序が計算に影響を与えるかどうかを意味します。一部のソース (ハッシュ・ベースのデータ・セットやマップ) では、出現順に意味はありません。ストリームの出現順に意味があるかどうかは、`ORDERED` ストリーム・フラグ

によって示されます。JDK コレクションの Spliterator は、このフラグをコレクションの仕様に基づいて設定します。中間処理の中には、`ORDERED` を注入する処理 (`sorted()`) や、クリアする処理 (`unordered()`) もあります。

ストリームに出現順がある場合、ほとんどのストリーム処理では、ストリームの出現順に従う必要があります。順次実行の場合、要素は出現した順のまま処理されるため、出現順を維持するかどうかは基本的に自由です。並列実行内であっても、多くの処理 (ステートレス中間処理や、`reduce()` などの特定の終端処理) では、出現順を維持しなくても実質的なコストは伴いません。ただし、その他の処理 (ステートフル中間処理や、セマンティックが出現順に結び付けられている `findFirst()` や `forEachOrdered()` などの終端処置) については、並列実行内で出現順に従わなければならないことが大きな意味を持ちます。ストリームの出現順が定義されている一方、結果ではその順序に意味がない場合、順序に影響される処理を含むパイプラインの並列実行は、`unordered()` 処理を使用して `ORDERED` フラグを削除することで高速化できる可能性があります。

出現順に影響される処理の一例として、ストリームを特定のサイズで切り捨てる `limit()` について考えてみましょう。順次実行内で `limit()` を実装するには、出現した要素の数のカウンターを保持し、指定の数に達した後はすべての要素を破棄するだけのことです。一方、並列処理の場合は、`limit()` の実装は遥かに複雑になります。それは、**最初**の `N` 個の要素を保持しなければならないためです。この要件は、並列処理を利用できるかどうかに大きく影響します。例えば、入力がセクションに分割される場合、あるセクションの結果が最終結果に含まれることになるかどうかは、そのセクションの前にあるすべてのセクションの処理が完了するまでわかりません。そのため通常、limit() の実装では、利用できるコアの一部しか利用しないとか、指定の長さに達するまで暫定的な結果全体をバッファーに入れるといった、誤った方法が採られます。

ストリームに出現順がなければ、`limit()` 処理では自由に**任意**の `N` 個の要素を選択できるため、実行を大幅に効率化することができます。出現順に意味がない場合、既知になった要素を即時にダウンストリームに送信できるため、バッファーは不要になります。スレッド間で必要な調整は、ターゲット・ストリームの長さを超えないようにするためのセマフォーだけです。

出現順のコストが伴い、さらに理解しにくい処理の例としては、ソートがあります。`sorted()` 処理では、出現順に意味がある場合は固定 (stable) ソート (同じ要素が入力内での出現順と同じ順序で出力に出現すること) を実装する一方、順不同のストリームには、(コストを伴う) 安定ソートは必要ありません。同様のシナリオは `distinct()` にもあります。ストリームに出現順がある場合、distinct() は、複数の同じ入力要素については**最初**の要素を送信する一方、順不同のストリームでは任意の要素を送信します。後者の場合も、大幅に効率的な並列実装にすることができます。

`collect()` を使用して集約する際も、同じような状況になります。順序付けされたストリームに対して `collect(groupingBy())` 処理を実行する場合、キーに対応する要素は、入力内で出現する順にダウンストリーム・コレクターに提示されなければなりません。多くの場合、この順序はアプリケーションにとっては意味がないため、任意の順で要素をコレクターに提示できます。その場合、並行コレクター (`groupingByConcurrent()` など) を選択するほうが好ましいこともあります。並行コレクターでは、出現順を無視して、すべてのスレッドが要素を共有並行データ構造体 (`ConcurrentHashMap` など) に直接収集することができます。この方法は、スレッドごとにそれぞれ

の中間マップに収集させた後、それらの中間マップをマージする方法と比べてコストがかかりません。

ストリームの作成

> “ストリームを分配するように既存のデータ構造体を改変するのは簡単です。”

JDK に含まれるクラスの多くは、ストリーム・ソースとしての役割を果たすよう改良されていますが、ストリームを分配するように既存のデータ構造体を改変させるのも簡単です。任意のデータ・ソースからストリームを作成するには、そのストリームの要素を対象とした `Spliterator` を作成し、その Spliterator を、最終的に生成されるストリームが順次または並列のどちらであるかを示す `boolean` フラグと併せて `StreamSupport.stream()` に渡します。

`Spliterator` 実装は、実装作業と、`Spliterator` をソースとして使用するストリーム・パイプラインのパフォーマンスの間でのトレードオフによって、品質が大幅に左右されます。`Spliterator` インターフェースには複数のメソッド (`trySplit()` など) があります。これらのメソッドは基本的にオプションです。分割処理を実装したくなければ、いつでも `trySplit()` から `null` を返すことができます。ただし、その場合、この `Spliterator` をソースとして使用するストリームは、並列処理を利用して計算を高速化できません。

Spliterator の品質に影響を与える考慮事項には、以下があります。

- Spliterator が正確なサイズをレポートするかどうか
- Spliterator で入力を分割することが可能かどうか
- 分割できる場合、入力をほぼ等分のセクションに分割できるかどうか
- 分割されるセクションのサイズを予測できるかどうか (`SUBSIZED` 特性によって反映できるかどうか)
- Spliterator が関連するすべての特性をレポートするかどうか

Spliterator を作成するのに最も簡単な方法は (ただし、結果的に品質は最も低くなります)、`Iterator` を `Spliterators.spliteratorUnknownSize()` に渡すことです。`Iterator` とサイズを `Spliterators.spliterator` に渡すと、それよりもわずかに品質の高い `Spliterator` になります。けれども、ストリームのパフォーマンス (特に並列パフォーマンス) が重要な場合は、該当するすべての特性を含め、完全な `Spliterator` インターフェースを実装する必要があります。`ArrayList`、`TreeSet`、`HashMap` などのコレクション・クラスの JDK ソースで提供している高品質の Spliterator の例は、独自のデータ構造体に応じてエミュレートすることができます。

第 3 回のまとめ

一般に、Streams は現状のままで十分なパフォーマンスを発揮しますが (対応する命令型コードよりパフォーマンスが優れていることもあります)、Streams が内部でどのように機能するのかを把握しておくと、このライブラリーを最大限有効に利用できるようになるだけでなく、カスタム・アダプターを作成して任意のデータ・ソースからストリームを作成できるようにもなります。シリーズ「Java Streams」の次の 2 回の記事では、並列処理について詳しく探ります。

関連トピック： java.util.stream のパッケージ・ドキュメント Javaによる関数型プログラミング — Java 8 ラムダ式と Stream (Venkat Subramaniam 著、オライリージャパン、2014) Mastering Lambdas: Java Programming in a Multicore World (Maurice Naftalin 著、McGraw-Hill Education、2014) Should I return a Collection or a Stream? RxJava ライブラリー

著者について

Brian Goetz

Brian Goetz is the Java Language Architect at Oracle and has been a professional software developer for nearly 30 years. He is the author of the best-selling book Java Concurrency In Practice (Addison-Wesley Professional, 2006).