Java Streams, Part 1: java.util.stream ライブラリー入門

コレクションや他のデータ・セットに対し、関数形式のクエリーを実行する

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、ラムダ式の威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、関数形式のクエリーを簡単に実行できるようにします。

このシリーズの他の記事を見る

Java SE 8 で導入された大きな言語機能としてラムダ式 (lambda expression) があります。ラムダ式は匿名メソッドであると見なすことができ、メソッドと同じように型付きパラメーター、本体、戻り値の型からなります。けれども本当に注目すべき点はラムダ式そのものではなく、ラムダ式によって何が可能になるかということでした。ラムダ式では**振る舞いをデータとして簡単に表現**できるため、より表現力豊かで強力なライブラリーを開発することができるのです。

そのようなライブラリーの 1 つが、同じく Java SE 8 で導入された `java.util.stream` (Streams) です。このライブラリーは、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を可能にします。以前の Java バージョンでも Streams のようなライブラリーを作成することはできたものの、データとしての振る舞いという簡潔な表現法がなかったため、作成したとしても、あまりにも厄介で誰も使いたがらなかったでしょう。Streams は、ラムダ式の威力を利用する Java 初のライブラリーであると見なせますが、機能はあくまで一般的です (ただし、コア JDK ライブラリーにしっかり統合されています)。Streams は、Java 言語の一部ではなく、より新しい言語機能を利用する、慎重に設計されたライブラリーです。

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どのようにして最大限活用するのかを説明します。

この記事は、`java.util.stream` の詳細を探るシリーズの第 1 回です。この記事では Streams ライブラリーを紹介し、このライブラリーの利点および設計原則について概説します。以降の記事では、ストリームを使用してデータをどのようにして集約および要約するのかを説明し、このライブラリーに含まれる要素やパフォーマンスの最適化を取り上げます。

ストリームによるクエリー

最も一般的なストリームの用途としては、コレクションに含まれるデータに対するクエリーを表現することが挙げられます。リスト 1 に、ストリーム・パイプラインの単純な例を示します。このパイプラインは、買い手と売り手の間で行われる購入取引をモデル化した一連のトランザクションを取り、ニューヨークの売り手による合計取引金額 (ドル) を計算します。

リスト 1. 単純なストリーム・パイプライン

```
int totalSalesFromNY
    = txns.stream()
          .filter(t -> t.getSeller().getAddr().getState().equals("NY"))
          .mapToInt(t -> t.getAmount())
          .sum();
```

> “ *Streams* は、合成という最も強力な計算法則を活用しています。 ”

`filter()` 処理では、ニューヨークの売り手によるトランザクションだけを選択します。`mapToInt()` 処理では、対象のトランザクションの取引金額を選択します。そして終端処理の `sum()` で、取引金額を合計します。

上記の例は簡潔で、読んで簡単に理解できますが、このクエリーは命令型 (`for`-loop) バージョンでも単純であり、数行のコード行を使用して表現できると指摘する反論もあるでしょう。けれども、ストリーム手法の利点は、比較的簡単な問題によっても明らかになります。Streams は、合成という最も強力な計算法則を活用しています。ストリームによるクエリーは、この法則に従って、単純なビルディング・ブロック (フィルタリング、マッピング、ソート、集約) を組み合わせて複雑な処理を構成します。そのため、問題が複雑化しても単純さを維持しやすく、同じデータ・ソースに対するアドホック型の計算よりも簡単に作成、理解することができるのです。

リスト 1 と同じ分野でのより複雑なクエリーとして、65 歳を超える買い手と取引した売り手を、名前を基準にソートして出力する場合を考えてみてください。このクエリーを旧式の方法 (命令型) で作成すると、リスト 2 に記載するようなクエリーになります。

リスト 2. コレクションに対するアドホック・クエリー

```
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Seller>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

上記のクエリーは、前のクエリーよりほんのわずかに複雑なだけですが、命令型の手法で作成したコードでは、編成の単純さと読みやすさは、すでに崩れ始めています。このコードを読むときに最初に目にするのは、計算の始まりでも終わりでもなく、最終的に不要になる中間結果の宣言です。このようなコードは、多くのコンテキストを頭に叩き込んでからでないと、コードが実際にどのような処理をするのかを理解できません。リスト 3 に、このクエリーを Streams を使用して作成する場合の例を記載します。

リスト 3. リスト 2 のクエリーを Streams を使用して表現する場合の例

```
txns.stream()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sorted(comparing(Seller::getName))
    .map(Seller::getName)
    .forEach(System.out::println);
```

リスト 3 のコードは、遥かに読みやすくなっています。それは、「ガーベッジ」変数 (`sellers` や `sorted` など) に気を散らされることがなく、コードを読む間、多くのコンテキストを把握している必要もないからです。このコードはまさに、問題文をほぼそのまま表現しています。読みやすいコードは、保守担当者が一目見ただけでその内容を正確に理解できるため、間違いを発生させにくいという特性もあります。

Streams のようなライブラリーで採用されている設計手法は、実用性の高い、関心の分離につながります。つまり、計算の「中身」を指定するのはクライアントの担当ですが、その「方法」はライブラリーが管理するということです。関心の分離によって、専門知識は並列的に分布しやすくなります。一般にクライアント・コードの作成者は問題領域をより深く理解しており、ライブラリーの作成者はコード実行のアルゴリズム的性質に関して豊富な専門知識を持っています。それぞれのこの類の関心の分離を可能にするライブラリーを作成する鍵は、データを渡すのと同じくらい簡単に振る舞いを渡せるようにすることにあります。振る舞いをデータとして渡すことができれば、呼び出し側で複雑な計算の構造を記述し、後はライブラリーに任せて実行ストラテジーを選択するといった API を実現できます。

ストリーム・パイプラインの詳細

ストリームによる計算の構造は共通しており、いずれもストリーム・ソース (stream source)、ゼロ個以上の中間処理 (intermediate operation)、そして単一の終端処理 (terminal operation) からな

ります。ストリームの要素は、オブジェクト参照 (`Stream<String>`) にすることも、プリミティブ型の int 値 (`IntStream`)、long 値 (`LongStream`)、double 値 (`DoubleStream`) にすることもできます。

Java プログラムが取り込むデータの大半はすでにコレクションに格納されているため、ストリームによる計算の多くでは、コレクションをソースとして使用します。JDK のすべての `Collection` 実装は、効率的なストリーム・ソースとして機能するよう、強化されています。その一方で、これ以外にも使用できるストリーム・ソースがあります。例えば、配列、ジェネレーター関数、組み込みファクトリー (数値範囲など) です。さらに、(このシリーズの第 3 回で説明するように) カスタム・ストリーム・アダプターを作成して、任意のデータ・ソースにストリーム・ソースの役割を持たせることも可能です。表 1 に、JDK に含まれるストリーム生成メソッドの一部を記載します。

表 1. JDK のストリーム・ソース

メソッド	説明
`Collection.stream()`	コレクションの要素からストリームを生成します。
`Stream.of(T...)`	ファクトリー・メソッドに渡された引数からストリームを生成します。
`Stream.of(T[])`	配列の要素からストリームを生成します。
`Stream.empty()`	空のストリームを生成します。
`Stream.iterate(T first, BinaryOperator<T> f)`	`first, f(first), f(f(first)) …`と続くシーケンスからなる無限ストリームを生成します。
`Stream.iterate(T first, Predicate<T> test, BinaryOperator<T> f)`	(Java 9 のみ) `Stream.iterate(T first, BinaryOperator<T> f)` と同様ですが、test 述部が `false` を返した最初の要素でストリームが終了するという点が異なります。
`Stream.generate(Supplier<T> f)`	ジェネレーター関数から無限ストリームを生成します。
`IntStream.range(lower, upper)`	下限値以上、上限値未満の要素からなる `IntStream` を生成します。
`IntStream.rangeClosed(lower, upper)`	下限値以上、上限値以下の要素からなる `IntStream` を生成します。
`BufferedReader.lines()`	`BufferedReader` の出力行からなるストリームを生成します。
`BitSet.stream()`	`BitSet` に含まれる設定ビットのインデックスからなる `IntStream` を生成します。
`Stream.chars()`	`String` に含まれる文字に対応する `IntStream` を生成します。

中間処理には、基準と一致する要素を選択する `filter()`、関数に応じて要素を変換する `map()`、重複を削除する `distinct()`、特定のサイズにストリームを切り捨てる `limit()`、そして `sorted()` などがあります。これらの処理は、ストリームを別のストリームに変換するためのものです。`mapToInt()` などの一部の中間処理は、あるタイプのストリームを取り、別のタイプのストリームを返します。リスト 1 の例では、ストリームは `Stream<Transaction>` として始まっていますが、後で `IntStream` に変わっています。表 2 に、中間ストリーム処理の一部を記載します。

表 2. 中間ストリーム処理

処理	内容
`filter(Predicate<T>)`	述部に一致するストリームの要素
`map(Function<T, U>)`	指定の関数をストリームの要素に適用した結果

`flatMap(Function<T, Stream<U>>`	ストリームを指定した関数を、そのストリームの要素に適用して生成されたストリームの要素
`distinct()`	重複が削除されたストリームの要素
`sorted()`	自然順でソートされたストリームの要素
`Sorted(Comparator<T>)`	指定のコンパレーターでソートされたストリームの要素
`limit(long)`	指定の長さに切り捨てられたストリームの要素
`skip(long)`	最初の N 個の要素が破棄されたストリームの要素
`takeWhile(Predicate<T>)`	(Java 9 のみ) 指定の述部が真でない最初の要素で切り捨てられたストリームの要素
`dropWhile(Predicate<T>)`	(Java 9 のみ) 指定の述部が真である要素の最初のセグメントが破棄されたストリームの要素

中間処理は常に遅延処理です。中間処理を呼び出すと、ストリーム・パイプラインの次のステージが準備されるだけです。中間処理の呼び出しによって何らかの処理が開始されることはありません。中間処理は、さらにステートレス処理とステートフル処理の 2 つのタイプに分かれます。ステートレス処理 (`filter()`、`map()` など) では各要素を別々に処理できます。これに対し、ステートフル処理 (`sorted()`、`distinct()` など) では、他の要素の処理に影響する、それまでにあった要素から状態を取り込むことができます。

リダクション (`sum()` または `max()`)、適用 (`orEach()`)、検索 (`findFirst()`) 処理などの終端処理が実行された時点で、データ・セットの処理が開始されます。終端処理は、結果を出すか、副次作用を与えるかのどちらかです。終端処理が実行されると、ストリーム・パイプラインは終了します。同じデータ・セットを再度トラバースする必要がある場合は、新しいストリーム・パイプラインをセットアップするという方法があります。表 3 に、終端ストリーム処理の一部を記載します。

表 3. 終端ストリーム処理

処理内容	説明
`forEach(Consumer<T> action)`	指定のアクションをストリームの各要素に適用します。
`toArray()`	ストリームの要素から配列を作成します。
`reduce`	ストリームの要素を集計値に集約します。
`collect(...)`	ストリームの要素を要約結果コンテナーに集約します。
`min(Comparator<T>)`	コンパレーターに応じてストリームの最小の要素を返します。
`max(Comparator<T>)`	コンパレーターに応じてストリームの最大の要素を返します。
`count()`	ストリームのサイズを返します。
`{any,all,none}Match(Predicate<T>)`	指定の述部にストリームのいずれかの要素またはすべての要素が一致するか、あるいはどの要素も一致しないかについての結果を返します。
`findFirst()`	ストリーム (存在する場合) の最初の要素を返します。
`findAny()`	ストリーム (存在する場合) の任意の要素を返します。

ストリームとコレクションの違い

ストリームは一見するとコレクションに似ているかもしれませんが (どちらもデータを含んでいると見なすことが可能)、実際のところ、この 2 つには大きな違いがあります。コレクションはデータ構造体です。その主な関心事はメモリー内のデータの編成であり、コレクションが存続するのは一定の期間に限られます。コレクションがストリーム・パイプラインのソースまたはターゲットとして使用されることもよくありますが、ストリームで中心になるのはデータではなく、**計算**です。任意のソース (コレクション、配列、ジェネレーター関数、または I/O チャネル) からのデータがパイプラインの計算ステップを経て処理されて、結果を出すか副次作用を与えるかした時点で、ストリームは終了します。ストリームは、処理する要素の格納場所にはなりません。ストリームのライフサイクルは、格納場所というよりも特定の時点 (つまり、終端処理が呼び出された時点) に近いと言えます。コレクションとは異なり、ストリームは無限にすることもできます。このため、一部の処理 (`limit()`、`findFirst()`) は短絡処理であり、有限計算によって無限ストリームを処理できます。

コレクションとストリームは、処理の実行方法という点でも異なります。コレクション上での処理は即時処理であり、データを変化させるものです。例えば、`List` に対して `remove()` メソッドが呼び出された場合、呼び出しが戻ると、リストの状態は指定の要素の削除を反映して変わっていることがわかります。ストリームの場合、即時に行われるのは終端処理だけで、他はすべて遅延処理です。ストリームによる処理は、データ・セットを変化させる処理ではなく、関数を適用して入力 (同じくストリーム) を変換することを意味します (ストリームをフィルタリングすると、入力ストリームの一部の要素からなる新しいストリームが生成されますが、フィルタリングによってソースから要素が削除されることはありません)。

ストリーム・パイプラインを一連の関数型変換として表現すると、遅延処理 (laziness)、短絡処理 (short circuiting)、処理融合 (operation fusion) などの有用な実行戦略を実現できます。短絡処理では、すべてのデータ・クエリーを検査することなくパイプラインを正常に終了することができます。例えば「1,000 ドルを超える最初の取引を検出」する場合、短絡処理を適用すれば、一致する取引が見つかった後は、他の取引を検査する必要がなくなります。一方、処理融合とは、データに対する複数の処理を単一パス内で実行できることを意味します。リスト 1 を例にとると、まず、一致する取引をすべて選択し、次に、対応する取引金額をすべて選択し、その後すべての金額を合計するという 3 つの処理を単一のパスに結合してデータに対して実行するということです。

リスト 1 とリスト 3 のような命令型バージョンのクエリーでは、中間計算結果 (フィルタリングやマッピングなどの結果) に応じてコレクションをマテリアライズするという方法を用いることがよくあります。これらの中間結果はコードを煩雑にするだけでなく、実行にも混乱を招くおそれがあります。中間コレクションのマテリアライズが役立つのは実装面だけで、結果には役立ちません。また、結局は破棄されるだけのデータ構造体に中間結果を編成するために、計算サイクルを無駄に消費することになります。

それとは対照的に、ストリーム・パイプラインはデータに対する処理をできるだけ数少ないパス、多くの場合は 1 つのパスに融合します (ただし、ソートなどのステートフルな中間処理は、複数パスの実行を必要とする障害点を作り出す可能性があります)。ストリーム・パイプラインの各ステージで、そのステージの要素が遅延して生成され、必要な場合にだけ要素が計算されて次のステージに直接渡されます。フィルタリングやマッピングの中間結果を保持するためのコレク

ションは必要にならないことから、中間コレクションの入力 (およびガーベッジ・コレクション) に費やす労力が省かれます。また、「横型」実行戦略 (単一のデータ要素のパスを、パイプライン全体を通して追跡すること) に従うのではなく、「縦型」実行戦略に従うと、処理対象のデータがキャッシュ内で「利用可能」になる頻度が多くなるため、計算に費やす時間が増し、データを待機する時間を短縮できます。

ストリームは計算に使用するだけでなく、これまで配列やコレクションを返していた API メソッドから集約を返すためにも使用することをお勧めします。API からストリームを返せば、すべてのデータを新しい配列やコレクションをコピーする必要がなくなるため、この方法のほうが通常は効率的です。ストリームを返すという方法が、柔軟性の向上につながることも珍しくありません。コレクションを返すためにライブラリーが選ぶコレクションの形態は呼び出し側が必要とするタイプではない場合がありますが、ストリームであれば任意のタイプのコレクションに簡単に変換できます (ストリームを返すのは適切ではなく、マテリアライズしたコレクションを返したほうが有効な主な事例は、呼び出し側がある時点での一貫したスナップショットを参照する必要がある場合です)。

並列処理

関数型変換として計算を構成することによってもたらされるメリットの 1 つは、コードに最小限の変更を加えるだけで、順次実行と並列実行を簡単に切り替えられることです。それは、ストリームによる計算を順次実行する場合と並列実行する場合の表現は、ほぼ同じであるためです。リスト 4 に、リスト 1 のクエリーをどのようにして並列実行するのかを示します。

リスト 4. リスト 1 の並列バージョン

```
int totalSalesFromNY
    = txns.parallelStream()
          .filter(t -> t.getSeller().getAddr().getState().equals("NY"))
          .mapToInt(t -> t.getAmount())
          .sum();
```

> “ストリーム・パイプラインを一連の関数型変換として表現すると、遅延処理 (laziness)、並列処理 (parallelism)、短絡処理 *(short* circuiting)、処理融合 *(operation fusion)* などの有用な実行戦略を実現できます。”

リスト 1 との違いは、順次ストリームではなく並列ストリームを要求している最初の行だけです。これは、Streams ライブラリーでは、これを実行する戦略から、計算の記述と構成を効果的に除外するため、他に必要となる変更はありません。これまでは、コードを並列コードに変更するためには完全な再作成が必要でした。その再作成には経費がかかるだけでなく、変更後の並列コードは順次バージョンと少しも似ていないため、エラーの原因にもなりがちです。

ストリーム処理は順次でも並列でも実行できますが、並列処理にさえすればパフォーマンスが向上するわけではないことに留意してください。並列実行は順次実行に比べ、処理速度が向上することも、変わらないことも、遅くなることもあります。最初は順次ストリームを使用しておき、並列処理によって高速化を図れる (そしてメリットがある) ことがわかってから、並列処理を適用するのが最善です。このシリーズの以降の記事で、並列処理を再び取り上げ、ストリーム・パイプラインの並列パフォーマンスを分析します。

ただし書き

Streams ライブラリーは計算のオーケストレーションを行いますが、計算を実行するには、クライアントが提供するラムダ式に対するコールバックが必要です。これらのラムダ式で実行できる処理には、特定の制約が適用されます。これらの制約に違反すると、ストリーム・パイプラインが失敗したり、誤った結果を計算したりする可能性があります。その上、副次作用を伴うラムダ式の場合、その副次作用のタイミング (または存在) が予想外のものとなったりすることもあります。

ほとんどのストリーム処理に渡すラムダ式は、非干渉であること、ステートレスであることが要件となります。非干渉とは、ストリーム・ソースを変更しないことを意味し、ステートレスとは、ストリーム処理の実行期間中に変化する可能性のある状態を使用 (読み取り/書き込み) しないことを意味します。また、リダクション処理 (例えば、`sum`、`min`、`max` などの要約データの計算) に渡すラムダ式は、結合的である必要もあります (または同様の要件に従っている必要があります)。

これらの要件がなぜ適用されるのかというと、その理由の一部は、パイプラインが並列で実行されると、ストリーム・ライブラリーが複数のスレッドから、データ・ソースにアクセスしたり、ラムダ式を同時に呼び出したりする可能性があるという事実に基づきます。このような要件によって制約しなければ、計算が正しく行われることを確実化できません (これらの制約により、並列処理とは関係なく、より単純で、より理解しやすいコードになる傾向もあります)。特定のパイプラインが並列で実行されることはないはずなので、これらの制約は無視できると思い込みたいでしょうが、その誘惑は抑えることが賢明です。そうしないと、コードに時限爆弾を埋め込むことになってしまいます。実行戦略が何であれ、正しく実行されるストリーム・パイプラインを表現するよう努めてください。

並行処理に関するあらゆるリスクの根源は、可変の状態が共有されることにあります。そのような状態を作り出す原因として考えられるのは、ストリーム・ソースです。ソースが `ArrayList` など従来型のコレクションの場合、Streams ライブラリーはストリーム処理の過程でそのソースが変更されないことを前提とします (明示的に同時アクセスが意図されたコレクション (`ConcurrentHashMap` など) は、この前提から除外されます)。非干渉の要件に従って、ストリーム処理中に他のスレッドによって変更されるソースを排除するだけでなく、ストリーム処理自体に渡されるラムダ式もソースを変更しないようにしなければなりません。

ストリーム処理自体に渡されるラムダ式は、ストリーム・ソースを変更しないことに加え、ステートレスである必要もあります。例えば、リスト 5 のコードは、先行する要素と重複する要素を排除しようとするため、このルールに違反しています。

リスト 5. ステートフルなラムダ式を使用するストリーム・パイプライン (誤った例です！)

```
HashSet<Integer> twiceSeen = new HashSet<>();
int[] result
    = elements.stream()
              .filter(e -> {
                  twiceSeen.add(e * 2);
                  return twiceSeen.contains(e);
              })
              .toArray();
```

このパイプラインが並列実行されると、2 つの理由から誤った結果になります。理由の 1 つとして、`twiceSeen` セットは、何の調整もなく複数のスレッドからアクセスされるため、スレッド・セーフではありません。もう 1 つの理由としては、データが区分化されていることから、特定の要素が処理されるときに、その要素に先行するすべての要素が処理済みであることが保証されません。

最善なのは、ストリーム処理に渡されるラムダ式には、まったく副次作用がないこと、つまりストリーム処理の実行中にヒープ・ベースの状態を変更したり、I/O を実行したりしないことです。副次作用があるとしたら、その副次作用がスレッド・セーフであることを確実にするために必要な調整をとるのは、ラムダ式を提供する側の責任になります。

さらに、すべての副次作用が実行されることさえ保証されません。例えば、リスト 6 では、ライブラリーが自由に、`map()` に渡されるラムダ式の実行をまったく実行しないことを選べます。ソースのサイズは既知であり、`map()` 処理がサイズを維持することは既知であり、そしてマッピングは計算の結果に影響しないことから、ライブラリーはマッピングをまったく実行しないことで、計算を最適化できます (この最適化では、計算を O(n) から O(1) に変えられるだけでなく、マッピング関数の呼び出しに伴う作業を排除できます)。

リスト 6. 副次作用が実行されない場合もあるストリーム・パイプライン

```
int count =
    anArrayList.stream()
               .map(e -> { System.out.println("Saw " + e); e })
               .count();
```

この最適化の (計算が大幅に高速化されること以外の) 影響に気付くのは、`map()` に渡されるラムダ式に副次作用がある場合のみです。その場合、それらの副次作用が生じないことに驚くでしょう。このように最適化できるのは、ストリーム処理は関数型変換であるという前提に基づきます。大抵、私たちは労力をかけずに、ライブラリーによってコードの実行を高速化できることに喜びます。このような最適化が可能になることに伴う犠牲は、ストリーム処理に渡すラムダ式の処理内容に関する制約、そして副次作用への依存に関する制約を受け入れなければならないことです (全体的に見れば、これは私たちにとって、かなり有利な引き換え条件です)

第 1 回のまとめ

`java.util.stream` ライブラリーは、コレクション、配列、ジェネレーター関数、範囲、あるいはカスタム・データ構造体などのさまざまなデータ・ソースに対するクエリーを、おそらく並列でも実行できる、関数型スタイルのクエリーとして単純かつ柔軟に表現する手段を提供します。一度このライブラリーを使い始めたら、二度と手離せなくなるはずです！次回は、Streams ライブラリーの最も強力な機能の 1 つとして、集約を取り上げます。

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).