Java Streams, Part 2: Streams を使用した集約

データを簡単に切り分ける

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。今回の記事では、`java.util.stream` パッケージを使用して、どのようにして効率的にデータを集約および要約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「Java Streams」シリーズの第 1 回では、Java SE 8 で新しく導入された `java.util.stream` ライブラリーについて紹介しました。第 2 回となるこの記事では、Streams ライブラリーの最も重要かつ柔軟な側面の 1 つに注目します。それは、データの集約および要約機能です。

アキュムレーター・アンチパターン

第 1 回で最初に取り上げた例では、Streams を利用して単純な集計処理を実行しました (リスト 1 を参照)。

リスト 1. Streams を利用し、宣言型を使用して集約値を計算する例

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

リスト 2 に、上記の計算を「従来の方法」を使用して作成した場合の例を記載します。

リスト 2. 同じ集約値を、命令型を使用して計算する例

```
int sum = 0;
for (Txn t : txns) {
    if (t.getSeller().getAddr().getState().equals("NY"))
        sum += t.getAmount();
}
```

第 1 回で説明したように、新しい方法を使用して作成した例は従来の方法よりコードが長いとはいえ、以下の理由から、新しい方法のほうがお勧めです。

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最大限活用するにはどのようするのかを説明します。

- 新しい方法では、コードを単純な複数の処理に分解して構成するため、コードが簡潔になります。
- コードは命令型 (どのようにして結果を計算するのかを指定する段階的なプロシージャー) ではなく、宣言型 (目的とする結果を宣言すること) で表現されます。
- 表現するクエリーが複雑になってきたとしても、この手法で作成されたコードは簡単にスケーリングできます。

集約という具体的な事例の場合、上記の他にも、この手法のほうが望ましい理由があります。リスト 2 は、「アキュムレーター・アンチパターン」の一例を表しています。具体的には、コードで最初に可変のアキュムレーター変数 (`sum`) を宣言し、それを初期化してからアキュムレーターをループ内で更新するというパターンです。これがなぜ悪い方法なのかと言うと、第一に、このスタイルのコードは並列化するのが困難です。同期するなどの調整がとられていないため、アキュムレーターへのあらゆるアクセスが、データ競合をもたらします (調整がとられているとしても、調整によって生じるコンテンションが、並列化から得られる効率性のメリットを台無しにするでしょう)。

このようにアキュムレーターを使用する手法が望ましくない理由のもう 1 つは、この手法はあまりにも下位レベル、つまりデータ・セット全体ではなく、個々の要素レベルでの計算をモデル化しているためです。「取引金額をひとつひとつ順番に繰り返し処理し、ゼロに初期化したアキュムレーターに各金額を追加する」というステートメントより、「すべての取引金額の合計」というステートメントのほうが、全体が抽象化されて、より直接的です。

命令型のアキュムレーターが誤ったツールだとしたら、何が正しいツールになるのでしょうか？その答えのヒントは、最初の例ですでに目にしている `sum()` メソッドにあります。けれどもこれは、リダクション (reduction) という強力で一般的な手法の特別な例にすぎません。リダクションは、単純かつ柔軟で、並列化できる処理であり、命令型の累積より上位レベルの抽象化で動作します。

リダクション

> “ リダクションは、単純かつ柔軟で、並列化できる処理であり、命令型の累積より上位レベルの抽象化で動作します。 ”

リダクション (「折りたたみ (folding)」とも呼ばれています) は、多種多様な累積処理を抽象化する、関数型プログラミングから生まれた手法です。例えば、タイプ T の要素 x_1、x_2、... x_n からなる空ではないシーケンス X と、T に対するバイナリー演算子 (ここでは * で表現) があるとすると、* を適用した X のリダクションは以下のように定義されます。

$$(x_1 * x_2 * ... * x_n)$$

通常の加算をバイナリー演算子として使用する数値のシーケンスに適用されるリダクションは、単なる合計にすぎません。けれども、リダクションの手法を使って記述できる処理は他にも多数あります。バイナリー演算子が「2 つの要素のうち、大きいほうの要素を取る」となっている場合 (これは、Java 内ではラムダ式 `(x,y) -> Math.max(x,y)` で表現できます。あるいはさらに単純な方法として、メソッド参照 `Math::max` で表現することもできます)、リダクション処理は、最大の値を見つけるという処理に対応します。

累積を、アキュムレーター・アンチパターンを使用して記述するのではなくリダクションとして記述すると、より**抽象的**かつ**簡潔**に、しかも**並列化**にも対応できるように計算を記述できます。ただし、それにはバイナリー演算子が結合性 (associativity) という単純な条件を満たすことが前提となります。以下の式の場合、要素 a、b、c について、バイナリー演算子 * は結合性を持ちます。

```
((a * b) * c) = (a * (b * c))
```

結合性とは、**グループ化は重要ではない**ことを意味します。バイナリー演算子に結合性があれば、任意の順序で問題なくリダクション処理を実行できます。順次実行内での自然な実行順は左から右への順です。並列実行内では、データは複数のセグメントに分割されて、各セグメントが個々に集約され、それらの結果が結合されます。この 2 つの手法が同じ結果を出すことは、結合性によって確実になります。このことは、結合性の定義を 4 つの項に展開させると理解しやすくなります。

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

上記の式の左側は、典型的な順次計算に相当します。右側は、入力シーケンスを複数のセクションに分割して、各セクションを並列して集約し、それらの結果を * を使用して結合するという、

並列実行に特有の区分化したセクションの実行に相当します (意外に思われるかもしれませんが、* は可換である必要はありません。ただし、リダクション用に一般的に使用される多くの演算子 (加算や最大など) は、可換でなければなりません。結合性があり、可換ではないバイナリー演算子の一例は、文字列連結です)。

Streams ライブラリーには、以下を含め、リダクションを目的とした複数のメソッドがあります。

```
Optional<T> reduce(BinaryOperator<T> op)
T reduce(T identity, BinaryOperator<T> op)
```

これらのメソッドの単純な形は、結合性バイナリー演算子を取り、その演算子を適用したストリーム要素のリダクションを計算するというものです。リダクションの結果は `Optional` として記述されます。入力ストリームが空の場合、リダクションの結果も同じく空になります (入力に 1 つの要素しかない場合、リダクションの結果はその要素です)。文字列の集合があるとしたら、以下のようにして、それらの要素の連結を計算できます。

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

上記の 2 つのメソッドの 2 つ目の形を使用する場合は、ID 値を指定します。ストリームが空の場合は、この値が結果としても使用されます。ID 値は、すべての x に対して以下の制約を満たす必要があります。

identity * x = x
x * identity = x

すべてのバイナリー演算子に ID 値があるわけではありません。ID 値があると、目的の結果を得られない場合もあります。例えば最大値を計算するとしたら、ID 値として `Integer.MIN_VALUE` を使用しようと考えるかもしれません (この値は要件を満たします)。けれども、その ID が空のストリームに対して使用される場合、空の入力と `Integer.MIN_VALUE` だけが含まれる入力との違いを区別することができなければ、求める結果が出るとは限りません (この違いは問題にならない場合もあれば、問題になる場合もあります。このことから、Streams ライブラリーでは ID を指定するかどうかをクライアントに任せています)。

文字列連結の場合、ID は空の文字列になるため、前の例は以下のように作り直すことができます

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

同様に、配列での整数の合計は以下のように記述できます

```
int sum = Stream.of(ints).reduce(0, (x,y) -> x+y);
```

(ただし実際には、`IntStream.sum()` コンビニエンス・メソッドを使用することになります)。

リダクションは整数や文字列以外にも適用できます。要素のシーケンスをそのタイプの単一の要素に集約する必要がある場合は、常にリダクションを適用できます。例えば、リダクションによって最も身長が高い個人を計算するとしたら、以下のように記述できます。

```
Comparator<Person> byHeight = Comparators.comparingInt(Person::getHeight);
BinaryOperator<Person> tallerOf = BinaryOperator.maxBy(byHeight);
Optional<Person> tallest = people.stream().reduce(tallerOf);
```

指定した演算子に結合性がない場合、または指定した ID 値が実際にはバイナリー演算子の ID ではない場合、処理が並列実行されると誤った結果になり、同じデータに対するそれぞれの実行結果が異なってくる可能性があります。

可変リダクション

リダクションでは値のシーケンスを取り、それを集約してシーケンスの合計や最大の値などといった結果を出します。その一方、単一の要約値に集約することが目的ではなく、結果を `List` や `Map` などのデータ構造体に編成しなければならない場合や、複数の要約値に集約しなければならない場合もあります。そのような場合は、`collect` という、`reduce` の可変バージョンを使用してください。

単純な例として、複数の要素を `List` に累積する場合を考えてみましょう。アキュムレーター・アンチパターンを使用するとしたら、以下のようなコードを作成することになります。

```
ArrayList<String> list = new ArrayList<>();
for (Person p : people)
    list.add(p.toString());
```

アキュムレーター変数が単純な値である場合はリダクションが累積に代わる有効な手法になりますが、このことは、アキュムレーターの結果がより複雑なデータ構造体になる場合にも当てはまります。リダクションのビルディング・ブロックは、ID 値と、2 つの値を新しい値に結合する手段です。可変リダクション・バージョンの場合、ビルディング・ブロックは以下のようになります。

- 空の結果コンテナーを生成する手段
- 新しい要素を結果コンテナーに取り込む手段
- 2 つの結果コンテナーをマージする手段

これらのビルディング・ブロックは、関数として簡単に表現することができます。上記の 3 番目の手段に相当する関数では、可変リダクション処理を並列実行することが可能です。つまり、データ・セットを複数のセクションに分割し、各セクションの累積を中間結果として生成した後、それらの中間結果をマージすることができます。このすべての処理を引き受けるのが、Streams ライブラリーに含まれる `collect()` メソッドです。

```
<R> collect(Supplier<R> resultSupplier,
            BiConsumer<R, T> accumulator,
            BiConsumer<R, R> combiner)
```

前のセクションで、リダクションを使用して文字列連結を計算する例を記載しました。その例のイディオムは正しい結果を導きますが、(Java 内では文字列は不変であり、文字列連結では文字列全体をコピーしなければならないため) 実行時間が $O(n^2)$ になります (つまり、一部の文字列が複数回コピーされるということです)。文字列連結をより効率的に表現するには、文字列を `StringBuilder` に収集するという方法があります。

```
StringBuilder concat = strings.stream()
                              .collect(() -> new StringBuilder(),
                                       (sb, s) -> sb.append(s),
                                       (sb, sb2) -> sb.append(sb2));
```

上記の手法では `StringBuilder` を結果コンテナーとして使用しています。`collect()` に渡される 3 つの関数は、デフォルト・コンストラクターを使用して空のコンテナーを作成し、`append(String)` メソッドでそのコンテナーに要素を追加して、`append(StringBuilder)` メソッドで 2 つの結果コンテナーをマージします。このコードは、ラムダ式ではなく、メソッド参照を使って表現したほうが簡潔になるでしょう。

```
StringBuilder concat = strings.stream()
                              .collect(StringBuilder::new,
                                       StringBuilder::append,
                                       StringBuilder::append);
```

ストリームを `HashSet` に収集する場合も同じようにして、以下のようなコードを作成することができます。

```
Set<String> uniqueStrings = strings.stream()
                                   .collect(HashSet::new,
                                            HashSet::add,
                                            HashSet::addAll);
```

上記のバージョンでは、結果コンテナーとして `StringBuilder` ではなく `HashSet` が使用されていますが、手法の点では変わりありません。同じくデフォルト・コンテナーを使用して新しい結果コンテナーを作成し、`add()` メソッドでそのコンテナーに新規要素を取り込み、`addAll()` メソッドで 2 つのコンテナーをマージしています。このコードを他の任意の種類のコレクションにどのように適用するのかを理解するのは簡単です。

可変の結果コンテナー (`StringBuilder` または `HashSet`) が使用されていることから、このバージョンはアキュムレーター・アンチパターンの一例だと考えるかもしれませんが、そうではありません。この例のアキュムレーター・アンチパターン・バージョンは、以下のようになります。

```
Set<String> set = new HashSet<>();
strings.stream().forEach(s -> set.add(s));
```

> “ 複数のコレクターを *1* つに組み合わせて構成することで、より複雑な集約にも対処できます。 ”

結合関数に結合性があり、副次作用に相いれないものがなければ、リダクションによって問題なくコードを並列化できます。それと同じく、`Stream.collect()` を使用した可変リダクションでも、特定の単純な整合性の要件 (`collect()` の仕様に概説されています) を満たすとすれば、問題なくコードを並列化することができます。主な違いは、`forEach()` バージョンを使用する場合は、複数のスレッドが単一の結果コンテナーに同時にアクセスしようとする一方、並列 `collect()` バージョンを使用する場合は、各スレッドに固有のローカル結果コンテナーがあり、これらのコンテナーの結果が後でマージされるという点です。

コレクター

`collect()` に渡される 3 つの関数 (結果コンテナーの作成、結果コンテナーへの結果の取り込み、結果コンテナーのマージ) の間の関係は非常に重要であることから、`Collector` という固有の抽象化および対応する単純化された `collect()` バージョンが提供されています。文字列連結の例は、Collector を使用して以下のように作成し直すことができます。

```
String concat = strings.stream().collect(Collectors.joining());
```

セットへの収集の例は、以下のように再作成できます。

```
Set<String> uniqueStrings = strings.stream().collect(Collectors.toSet());
```

`Collectors` クラスには、コレクションへの累積、文字列連結、リダクションやその他の集約計算、そして (`groupingBy()` を使用した) 要約表の作成など、多くの一般的な集約処理のファクトリーが含まれています。表 1 に、組み込みコレクターの一部を記載します。これらのコレクターで十分でなければ、独自のコレクターを作成することも簡単です (「カスタム・コレクター」セクションを参照)。

表 1. 組み込みコレクター

コレクター	動作
`toList()`	要素を `List` に収集します。
`toSet()`	要素を `Set` に収集します。
`toCollection(Supplier<Collection>)`	要素を特定のタイプの `Collection` に収集します。
`toMap(Function<T, K>, Function<T, V>)`	要素を `Map` に収集し、指定のマッピング関数を適用して、それらの要素をキーと値に変換します。
`summingInt(ToIntFunction<T>)`	`int` 値を指定したマッピング関数を各要素に適用した結果を合計します (`long` 値および `double` 値のバージョンもあります)。
`summarizingInt(ToIntFunction<T>)`	`int` 値を指定したマッピング関数を各要素に適用した結果の合計、最小値、最大値、数、平均を計算します (`long` 値および `double` 値のバージョンもあります)。
`reducing()`	リダクション処理を要素に適用します (通常は、`groupingBy` などと併せて、ダウンストリーム・コレクターとして使用されます。各種バージョンがあります)。
`partitioningBy(Predicate<T>)`	指定の述部に該当するかどうかを基準に、要素を 2 つのグループに分けます。
`partitioningBy(Predicate<T>, Collector)`	要素をパーティションに分割し、指定のダウンストリーム・コレクターを使用して、各パーティションを処理します。
`groupingBy(Function<T,U>)`	ストリームの要素に適用する指定の関数をキーとして、そのキーを共有する要素のリストを値として持つ `Map` に要素をグループ化します。
`groupingBy(Function<T,U>, Collector)`	要素をグループ化し、指定のダウンストリーム・コレクターを使用して、各グループに関連付けられた値を処理します。
`minBy(BinaryOperator<T>)`	要素の最小値を計算します (最大値を計算する `maxBy()` もあります。)
`mapping(Function<T,U>, Collector)`	指定のマッピング関数を各要素に適用し、指定のダウンストリーム・コレクターを使用して処理します (通常は、`groupingBy` などと併せて、ダウンストリーム・コレクター自体として使用されます)。
`joining()`	要素が `String` 型であることを前提に、要素を文字列に結合します。場合によっては区切り文字、プレフィックス、サフィックスを使用します。
`counting()`	要素の数を計算します (通常は、ダウンストリーム・コレクターとして使用されます)。

複数のコレクター関数をまとめて `Collector` に抽象化することで、構文が単純になります。けれども、その真のメリットは、`groupingBy()` コレクターを使用して要素から派生したキーに応じて要素を `Map` に収集するといった複雑な集計に対処するために、コレクターを 1 つに組み合わせて構成する場合に明らかになります。例えば、売り手をキーとした 1,000 ドルを超える取引の `Map` を作成するには、以下のコードを作成します。

```
Map<Seller, List<Txn>> bigTxnsBySeller =
    txns.stream()
        .filter(t -> t.getAmount() > 1000)
        .collect(groupingBy(Txn::getSeller));
```

一方、売り手ごとの取引の `List` ではなく、売り手ごとの最大取引が必要だとします。売り手ごとの結果をキーにすることに変わりはありませんが、各売り手に関連付けられた取引をさらに処理して、最大取引に集約する必要があります。その場合、キーごとの要素をリストに収集して、そのリストを別のコレクター (ダウンストリーム・コレクター) に渡すのではなく、`groupingBy()`

の別のバージョンを使用するという方法があります。ダウンストリーム・コレクターとしては、`maxBy()` などのリダクション処理を選択できます。

```
Map<Seller, Txn> biggestTxnBySeller =
    txns.stream()
        .collect(groupingBy(Txn::getSeller,
                            maxBy(comparing(Txn::getAmount))));
```

上記では、売り手をキーとしたマップに取引をグループ化しますが、そのマップの値は、`maxBy()` コレクターを使用して収集した売り手別の売り上げをすべての収集した結果です。売り手ごとの最大取引ではなく、合計取引金額が必要だとしたら、`summingInt()` コレクターを使用できます。

```
Map<Seller, Integer> salesBySeller =
    txns.stream()
         .collect(groupingBy(Txn::getSeller,
                            summingInt(Txn::getAmount)));
```

地域別、売り手別などといった複数レベルの要約を取得する場合は、単に別の `groupingBy` コレクターをダウンストリーム・コレクターとして使用するだけで対処できます。

```
Map<Region, Map<Seller, Integer>> salesByRegionAndSeller =
    txns.stream()
        .collect(groupingBy(Txn::getRegion,
                            groupingBy(Txn::getSeller,
                                       summingInt(Txn::getAmount))));
```

別の分野での例として、ドキュメント内での単語の出現回数のヒストグラムを計算する場合を考えてみましょう。この場合、`BufferedReader.lines()` を使用してドキュメントを行に分け、`Pattern.splitAsStream()` を使用して各行を単語のストリームに分割した後、`collect()` と `groupingBy()` を使用して、単語をキーとし、キーの値を該当する単語の数とする `Map` を作成するという方法を採ることができます (リスト 3 を参照)。

リスト 3. Streams を使用して単語数のヒストグラムを計算する例

```
    Pattern whitespace = Pattern.compile("\\s+");
    Map<String, Integer> wordFrequencies =
        reader.lines()
              .flatMap(s -> whitespace.splitAsStream())
              .collect(groupingBy(String::toLowerCase),
                                  Collectors.counting());
```

カスタム・コレクター

JDK に用意されている標準コレクターのセットはかなり充実していますが、独自のコレクターを作成するのも簡単です。リスト 4 に記載する、ごく単純な `Collector` インターフェースは、入力タイプ `T`、アキュムレーター・タイプ `A`、最終戻り値タイプ `R` の 3 つを使用してパラメーター化されています (`A` と `R` が同じであることはよくあります)。このインターフェースのメソッドは、前に説明した 3 つの引数を使用する `collect()` バージョンが受け入れる関数と同様の関数を返します。

リスト 4. `Collector` インターフェース

```
public interface Collector<T, A, R> {
    /** Return a function that creates a new empty result container */
    Supplier<A> supplier();

    /** Return a function that incorporates an element into a container */
    BiConsumer<A, T> accumulator();

    /** Return a function that merges two result containers */
    BinaryOperator<A> combiner();

    /** Return a function that converts the intermediate result container
        into the final representation */
    Function<A, R> finisher();

    /** Special characteristics of this collector */
    Set<Characteristics> characteristics();
}
```

`Collectors` 内でのコレクター・ファクトリーの実装は概して単純です。例えば、`toList()` の実装は以下のようになります。

```
return new CollectorImpl<>(ArrayList::new,
                           List::add,
                           (left, right) -> { left.addAll(right); return left; },
                           CH_ID);
```

上記の実装では `ArrayList` を結果コンテナーとして使用し、`add()` を使用して結果コンテナーに要素を取り込み、`addAll()` を使用してそのリストを別のリストにマージして、characteristics によって終了関数が identity 関数であることを示しています (これにより、ストリーム・フレームワークによって実行を最適化することが可能になります)。

前にも説明したように、整合性に関する要件がいくつかあります。これらの要件は、リダクションにおける ID とアキュムレーター関数との間に存在する制約に似ています。整合性についての要件は、`Collector` の仕様に概説されています。

もう少し複雑な例として、データ・セットに関する要約統計を作成するという問題を考えてみましょう。リダクションを使用すれば、数値データ・セットの合計、最小値、最大値、または数を簡単に計算できます (算出された合計と数から、平均も計算できます)。このすべての計算を単一のパス内でデータに対して実行するとなると、リダクションを使用して対処するのは難しくなってきますが、この計算を効率的に (お望みであれば並列で) 実行する `Collector` は簡単に作成することができます。

`Collectors` クラスに含まれる、`IntSummaryStatistics` を返す `collectingInt()` というファクトリー・メソッドは、まさにこの例に必要なメソッドです。このメソッドであれば、単一のパス内で合計、最小値、最大値、カウント数、平均のすべてを計算できます。`IntSummaryStatistics` の実装は単純なので、任意のデータ要約を計算する同じようなコレクターも簡単に作成できます (あるいは、既存の実装を拡張することもできます)。

リスト 5 に、`IntSummaryStatistics` クラスを示します。実際の実装は、この他の詳細 (要約統計を返すゲッターなど) も含まれていますが、実装の中核となっているのは単純な `accept()` メソッドと `combine()` メソッドです。

リスト 5. `summarizingInt()` コレクターによって使用される `IntSummaryStatistics` クラス

```
public class IntSummaryStatistics implements IntConsumer {
    private long count;
    private long sum;
    private int min = Integer.MAX_VALUE;
    private int max = Integer.MIN_VALUE;

    public void accept(int value) {
        ++count;
        sum += value;
        min = Math.min(min, value);
        max = Math.max(max, value);
    }

    public void combine(IntSummaryStatistics other) {
        count += other.count;
        sum += other.sum;
        min = Math.min(min, other.min);
        max = Math.max(max, other.max);
    }

    // plus getters for count, sum, min, max, and average
}
```

見てのとおり、このクラスはかなり単純なクラスです。新しいデータ要素が確認されるごとにどのようにして要約が更新されるのかも、`IntSummaryStatistics` ホルダーがどのようにして結合されるのかも、ひと目で理解できます。リスト 6 に記載する `Collectors.summarizingInt()` の実装も同じように単純です。この実装で作成している `Collector` は、整数値を指定したエクストラクター関数を適用し、その結果を `IntSummaryStatistics.accept()` に渡すことによって要素を取り込みます。

リスト 6. `summarizingInt()` コレクター・ファクトリー

```
public static <T>
Collector<T, ?, IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper) {
    return new CollectorImpl<T, IntSummaryStatistics, IntSummaryStatistics>(
            IntSummaryStatistics::new,
            (r, t) -> r.accept(mapper.applyAsInt(t)),
            (l, r) -> { l.combine(r); return l; },
            CH_ID);
}
```

(`groupingBy()` の例でのように) コレクターを簡単に 1 つに組み合わせて構成できること、そして新しいコレクターを簡単に作成できることの両方が、コードの簡潔さとわかりやすさを保ちつつ、ストリーム・データのほぼあらゆる要約を計算することを可能にしているのです。

第 2 回のまとめ

Streams ライブラリーに用意されている集約機能は、このライブラリーの最も有用で柔軟な側面の 1 つです。単純な値であれば、順次実行内でも並列実行内でも、リダクションを使用して簡単に集約することができます。複雑な要約を作成する場合には `collect()` を使用できます。このライブラリーには、基本的なコレクターのセットが同梱されていて、これらのコレクターを 1 つに組み合わせて構成することで、複雑な集約にも対処できるようになっています。また、このコレクターのセットに追加する独自のコレクターも簡単に作成できます。

第 3 回では、パフォーマンスが極めて重要な場合に Streams ライブラリーを最大限有効に利用するにはどうするのかを理解できるよう、このライブラリーの内部の仕組みを探ります。

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).