BIO

Ben Evans(@kitty-lyst) jClarity最高経営責任者兼London Java Community(LJC)主催者兼Java SE/EE Executive Committeenメンバー。

Martijn Verburg (@jClarity最高技術責任者兼LJC共同主催者。

Trisha Gee (@trisha_gee) MongoDBを提供する10gen開発者。LJCとGraduate Developer Communityに従事。

JOHN BLYTHE(EVANS氏、GEE氏)
BOB ADLER(VERBURG氏)

パート1

Java言語とJVMで使用するラムダ式の探究

ラムダ式を使用して、完結で真の並列実行が可能なコードを記述する

ラムダ(Lambda)式は、Java SE 8でもっとも待ち望まれている新機能のひとつです。専門家グループはこの新しい関数型プログラミング構造を「ラムダ式」と名付けました。ラムダ式は、クロージャ、関数リテラル、匿名関数、シングル抽象メソッド(SAM)などと呼ばれることもあります。これらの用語には若干意味の異なるものもありますが、すべての用語が同じ基本機能を表しています。

ラムダ式は最初のうちは見慣れないでしょうが、非常に習得しやすい構文です。最新型のマルチコアCPUを十分に活用できるアプリケーションを作成するためには、ラムダ式を使いこなすことが不可欠になります。

重要な概念として、ラムダ式はデータとして渡すことのできる小さな機能にすぎないということを念頭に置いておく必要があります。また、現在の順次処理的な外部イテレータとは異なる、コレクションの内部イテレータの実行方法について理解することが必要です。

本記事では、ラムダ式を使用する理由とラムダ式の使用例について説明し、当然ながらラムダ式の構文についても確認します。

ラムダ式が必要となる理由

プログラマがラムダ式を必要とする理由はおもに次の3つです。

- より簡潔なコードの記述
- 追加の機能を渡すことによるメソッドの変更
- マルチコア・プロセッシングのサポートの強化

より簡潔なコードの記述:ラムダ式は、関数を1つしか持たないJavaクラスをより整然と実装するための手法です。

たとえば、UIコードのリスナーやハンドラ、あるいは並列に実行されるCallableやRunnableオブジェクトなどを定義した、多数の匿名インナー・クラスがコードに含まれているとします。その場合、ラムダ式のスタイルに切り替えることで、より短く、理解しやすいコードになります。

メソッドの変更:実際に必要としている機能が既存のメソッドに存在しないことがあります。たとえば、Collectionインタフェースのcontains()メソッドは、渡されたオブジェクトそのものがコレクション内に存在する場合のみtrueを返します。このcontains()メソッドの機能を微調整する方法はありません。たとえば、検索する文字列とは大文字/小文字のみが異なる文字列が存在する場合もtrueを返すようにメソッドを変更することはできません。

簡単に言うと、ここで必要としていることは、既存のメソッドに「何らかの独自の新しいコードを渡して」、このコードを既存のコードから呼び出させることです。ラムダ式は、メソッドに渡してコールバックする必要があるコードを表現する優れた手法です。

マルチコア・プロセッシングのサポートの強化: 現代のCPUには複数のコアが搭載されています。そのため、マルチスレッド対応コードは、単一のCPU上で時分割方式で実行するのではなく、本当の意味で並列実行することが可能です。ラムダ式を使用した場合、Javaで関数型プログラミングのイディオムがサポートされるため、CPUの複数のコアを効果的に活用できるシンプルなコードを記述できます。

たとえば、CPU上で実行可能なすべてのハードウェア・スレッドを利用して、大規模なコレクションの操作を並列化できます。これには、フィルタ、マップ、リデュースといった、並列化に対応したイディオムを活用します。これらのイディオムについては後ほど取り上げます。

ラムダ式の概要

前述の大文字/小文字が異なる文字列を扱う例では、toLowerCase()というコード

表現を渡すことが必要です。このコードをcontains()メソッドのバリエーションに第2パラメータとして渡すためには、次の方法を考える必要があります。

- コードの断面を値（ある種のオブジェクト）のように扱う方法
- そのコード・ブロックを変数に代入する方法

言いかえれば、ロジックの一部を何らかでラップしてメソッドに渡す方法が必要です。この点を少し具体的に理解するために、既存のJavaコードをラムダ式を使用したコードに置き換える例をいくつか見ていきます。

フィルタ: メソッドに渡すコードの最適な例として、フィルタがあります。たとえば、Java SE 7よりも前のJava SEで、あるパスの内部に属するディレクトリを識別するためにjava.io.FileFilterを使用しているとします（**リスト1**参照）。ラムダ式を使用することで、リスト1のコードを**リスト2**のように大幅に簡略化できます。

型（FileFilter）は、代入文の左辺から推論されます。代入文の右辺は、FileFilterインタフェースのaccept()メソッドを非常に短くした表現のように見えます。つまり、Fileを引数に取り、f.isDirectory()を評価した結果のBooleanを返しています。

型推論は次のように動作します。まず、コンパイラは、FileFilterのメソッドがaccept()のみであるとわかっているため、リスト2のラムダ式は必ずaccept()メソッドの実装コードとなります。また、accept()の唯一のパラメータがFile型であることもわかっています。そのため、fは必ずFile型となります。型推論により簡略化したコードを**リスト3**に示します。このコードからわかるように、ラムダ式を使用すれば定型コードを大幅に削減できます。

ラムダ式に慣れると、ロジックの流れが格段に読み取りやすくなります。読み取りやすさを実現する重要なポイントのひとつは、フィルタ・ロジックを、そのロジックを必要としているメソッドのすぐ側に配置することです。

イベント・ハンドラ: UIコードもまた、匿名インナー・クラスのコード量が非常に多くなる領域のひとつです。ここでは、ボタンにクリック・リスナーを割り当てる例を取り上げます（**リスト4**参照）。

リスト4では、「ボタンが押されたときに、このメソッドを呼び出す」ためだけに、多くのコードを記述しています。ラムダ式を使用した場合、**リスト5**のようなコードを記述できます。

listenerオブジェクトは必要に応じて再利用できますが、一度しか使用しないのであれば、一般的に**リスト6**のようなコードが良いスタイルだと考えられています。

この例には見慣れない余分な中括弧{}が登場していますが、actionPerformedは戻り値のないvoid型メソッドであるため、このような構文が必要になっています。この中括弧の構文については後ほど確認します。

これ以降では、コレクションを処理する近代的なコードでラムダ式が果たす役割について見ていきます。特に、外部イテレータと内部イテレータという2つのプログラミング・スタイルの間の移行について取り上げます。

LISTING 1 LISTING 2 LISTING 3 LISTING 4 LISTING 5 LISTING 6

```
File dir = new File("/an/interesting/location/");
FileFilter directoryFilter = new FileFilter() {
    public boolean accept(File file) {
        return file.isDirectory();
    }
};
File[] directories = dir.listFiles(directoryFilter);
```

Download all listings in this issue as text

外部イテレータと内部イテレータの比較

これまで、Javaコレクションの標準的な処理には、外部イテレータを使用していました。外部イテレータと呼ばれる理由は、コレクションの外部に制御フローが存在することです。この制御フローを使用して、コレクション内の各要素に対する反復処理を実行します。外部イテレータは、従来よりコレクションの処理に使用されてきた方法です。外部イテレータという用語を知らない、あるいは使用していないとしても、手法そのものはほとんどのJavaプログラマにとって非常になじみのあるものです。

たとえば、拡張forループのような言語構造で外部イテレータを作成し、その外部イテレータ・オブジェクトを使用してコレクションをトラバースします（**リスト7**参照）。

このアプローチでは、コレクション・クラスは、コレクション内の全要素の「モノリシックな」（すなわち、連続した単一の）ビューです。このコレクション・オブジェクトでは、プログラマが必要とするコレクション内の任意の要素にランダムにアクセスできます。

このような外部イテレータのビューでは、コレクション・インスタンスのiterator()というメソッドを呼び出して反復処理を実行します。iterator()メソッドは、同じコレクションの要素をさらに限定したビューであるIterator型オブジェクトを返します。Iteratorはコレクション要素の単純な順次処理を目的としたオブジェクトであり、ランダム・アクセスを可能にするインタフェースは公開されていません。また、この順次処理の性質によって、コレクションを並列アクセスで扱おうとした場合には、悪名高いConcurrentModificationExceptionが発生します。

一方、コレクション・オブジェクトに、イテレータ（およびループ）を内部的に管理させるという方法もあります。

これが内部イテレータと呼ばれるアプローチであり、ラムダ式を使用する場合に適しています。

Project Lambdaでは、ラムダ式用に新しい構文が追加されるほか、コレクション・ライブラリのメジャー・アップグレードも実施されます。コレクションのアップグレードは、内部イテレータを使用するコードをより簡単に記述できるように、よく知られた関数型プログラミングのさまざまなイディオムをサポートすることを目的にしています。

ラムダ式の関数型イディオム

コレクションに次のような操作を1度または複数回実行する必要があることは、大半の開発者がさまざまな場面において経験しています。

- ある条件を満たさない要素をすべてフィルタによって除外し、新しいコレクションを作成すること
- コレクションの各要素を変換し、変換後のコレクションを使用すること
- コレクション全体の特性を表す総合的な値を計算すること（値の合計や平均の算出など）

これらのタスク（上から順に、フィルタ、マップ、リデュースと呼ばれるイディオム）には、ある重要な共通点があります。それは、これらのタスクにはコレクションの各要素に対して実行されるコードが必要だということです。

ある要素を検証して条件を満たしているかを確認するのか（フィルタ）、ある要素を新しいコレクションの新しい要素に変換するのか（マップ）、あるいは全体的な計算に使用する値を算出するのか（リデュース）にかかわらず、いずれの場合にも、「各要素に対して実行する必要のあるコード」が重要なテーマとなります。

これはつまり、内部イテレータで使用されるこういったコードを表現するシンプルな手法が必要であることを意味しています。幸いにも、Java SE 8ではこういった表現に適した言語要素が提供されます。

基本的な関数型イディオムに使用するJava SE 8のクラス

Java SE 8には、前述の基本的な関数型イディオムを実装することを目的としたクラスがあります。たとえば、Predicate、Mapper、Blockです。これらはその他のクラスとともに、java.util.functionsという新しいパッケージに含まれています。

それでは、Predicateの詳細を見ていきます。Predicateクラスは通常、フィルタ・イディオムの実装に使用します。Predicateクラスをコレクションに適用することで、フィルタ条件を満たす要素のみを含む新しいコレクションが返されます。

述語（Predicate）についてはさまざまな解釈が存在しますが、Java SE 8においては、述語とは、変数の値に応じてtrueまたはfalseの評価を返すメソッドを表します。

前述の例をもう一度取り上げます。すなわち、文字列のコレクションに対し、大文字/小文字は区別せずに、特定の文字列が含まれるかを確認します。

LISTING 7 LISTING 8 / LISTING 9 / LISTING 10

```
List<String> myStrings = getMyStrings();
for (String myString : myStrings) {
  if (myString.contains(possible))
   System.out.println(myString + " contains " + possible);
  }
}
```

Download all listings in this issue as text

Java SE 7では、外部イテレータを使用する必要があります（**リスト8**参照）。一方Java SE 8では、PredicateとCollections APIの新しいヘルパー・メソッド（filter）を使用して、コードを大幅に簡略化できます（**リスト9**参照）。また、一般的な関数型プログラミングのスタイルに従えば、**リスト9**のコードは**リスト10**のように一行で記述できます。ご覧のとおり、**リスト10**のコードは非常に読みやすく、しかも内部イテレータを使用するメリットもあります。

最後に、ラムダ式の完全な構文について詳細を説明します。

ラムダ式の構文ルール

ラムダ式の基本的な形式では、受け取ることのできるパラメータ・リストから書き始め、矢印（->）を挟んで、何らかのコード（ボディ）を記述します。

注: ラムダ式の構文は今後変更される可能性がありますが、本記事の執筆時点では、以降の例に示す構文は問題なく動作します。

ラムダ式では、型推論を多用しています。これは、他のJava構文と比較すると非常に珍しいことです。

先ほどの例を詳しく見てみます（**リスト11**参照）。ActionListenerの定義を見ると、このクラスがメソッドを1つだけ含むインタフェースであることがわかります（**リスト12**参照）。

そのため、**リスト11**のコードの右側にあるラムダ式は、「ActionListenerインタフェースで宣言されている唯一のメソッドに対するメソッド定義である」ことを容易に理解できます。なお、注意すべき点として、Javaの静的型付けに関する通常のルールには従う必要があります。従わない場合、型推論は正しく動作しません。

以上から、ラムダ式の使用により、以前であれば匿名インナー・クラスを使用していたコードを簡潔に記述できることがわかります。

構文について注目すべき点がもう1つあります。**リスト13**の例を見てください。一見、ActionListener

の例とほぼ同じようですが、ここでFileFilterインタフェースの定義を確認してみます（**リスト14**参照）。

FileFilterのaccept()メソッドはBoolean値を返していますが、明示的なreturn文はどこにもありません。その代わりに、戻り値の型が式（**リスト13**の場合、f.isDirectory()）から推論されます。f.isDirectory()の戻り値は言うまでもなく、Boolean値です。

戻り値を返さないvoid型メソッドの扱い方が少し異なる理由はここにあります。void型メソッドを扱う際は、ラムダ式のコードの一部を中括弧で囲むという追加の構文が使用されます。この構文を使用しなければ、型推論は正しく動作しません。ただし、この中括弧の構文は今後変更される可能性があることに注意してください。

ラムダ式のボディには複数の文を含めることができます。複数の文を含める場合は、ボディを括弧で囲む必要があります。また、「戻り値型の推論」の構文は動作しないため、必ずreturnキーワードを記述する必要があります。

最後に1つ注意する点があります。現段階ではラムダ式の構文をサポートしているIDEはほとんどないため、初めてラムダ式を試すときには、javacで生成されるコンパイラ警告に十分注意してください。

いつもきれいにラムダ式を使えば、関数を1つしか持たないJavaクラスを<u>よりすっきり実装できます</u>。.

LISTING 11 LISTING 12／LISTING 13／LISTING 14

```
ActionListener listener = event -> {ui.showSo-
mething();};
```

Download all listings in this issue as text

まとめ

ラムダ式は、Java SE 5で総称型が導入されて以来のJavaにおける大規模な新言語機能です。ラムダ式を適切に利用することで、より簡潔なコードの記述、追加の機能を渡すことによるメソッドの変更、マルチコア・プロセッサの有効活用が可能になります。本記事をここまで読んだ皆さんは、すぐにでもラムダ式を試したいと思われたことでしょう。

ラムダ式の機能が含まれたJava SE 8のスナップショット・ビルドは、Project Lambda公式ページから入手できます。バイナリを試すだけではなく、公式ページに掲載されている「State of the Lambda」の各記事もぜひお読みください。 **</article>**

LEARN MORE

- ラムダ式に関する疑問がある場合や、世界中で開催されているHack Dayイベントに参加するためには、Adopt OpenJDKプロジェクトにご参加ください