関数型の考え方: 遅延処理、第 1 回

Java での遅延評価を探る

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 12月 20日

多くの関数型プログラミング言語に共通する機能に「遅延評価」があります。遅延評価では、式を宣言時に評価するのではなく、必要になった時点でのみ評価します。Java ではこのようなスタイルの遅延処理をサポートしていませんが、いくつかのフレームワークと関連言語がサポートしています。この記事では、Pure Java と関数型フレームワークを使用して Java アプリケーションに遅延処理を統合する方法を説明します。

このシリーズの他の記事を見る

この連載について

この連載の目的は、読者の皆さんの考え方を関数型の発想へと方向転換し、よくある問題を新たな考え方で検討することによって、日常的なコーディングの改善方法を見つけるお手伝いをすることです。そのために、関数型プログラミングに特徴的な概念、関数型プログラミングを Java 言語で行えるようにするフレームワーク、JVM 上で動作する関数型プログラミング言語、そして今後、言語設計を学習する上での方向性などについて詳しく探ります。この連載の対象読者は、Java および Java の抽象化がどのように機能するかは知っていても、関数型言語を使用した経験がほとんど、あるいはまったくない開発者です。

「遅延評価」(式を評価するタイミングを可能な限り遅らせること) は、多くの関数型プログラミング言語に備わっている機能です。コレクションに含まれる要素を事前に計算する代わりに、遅延コレクションで必要に応じて要素を提供することには、いくつかの利点があります。第一に、コストのかかる計算を、その計算がどうしても必要になるまで遅らせることができます。第二に、要求を受け取り続ける限り、要素を提供し続ける無限コレクションを作成することができます。そして第三に、`map` や `filter` などの関数型の概念を遅延して使用することで、より効率的なコードを生成することができます (「参考文献」のリンクにアクセスして、この件に関する Brian Goetz 氏による説明を参照してください)。遅延処理は、Java ネイティブではサポートしていませんが、いくつかのフレームワークと Java を継承する言語がサポートしています。これらのフレームワークおよび言語について、今回の記事と次回の記事の 2 回にわたって探っていきます。

一例として、リストの長さを出力する以下の擬似コードのスニペットで考えてみましょう。

```
print length([2+1, 3*2, 1/0, 5-4])
```

このコードを実行してみると、これを作成したプログラミング言語のタイプが「正格」であるか「非正格」(非正格は「遅延」としても知られています) であるかによって、実行結果は異なってきます。正格なプログラミング言語でこのコードを実行すると (あるいは、おそらくコンパイルの時点から)、このリストの 3 番目の要素が原因で `DivByZero` 例外が発生します。非正格な言語では、結果は `4` になります。つまり、リスト内の項目数を正確に報告しているということです。結局のところ、上記で呼び出しているメソッドは `length()` であり、`lengthAndThrowExceptionWhenDivByZero()` ではありません！非正格言語として使用されている数少ない言語の 1 つに、Haskell があります (「参考文献」を参照)。また、Java では非正格評価をサポートしていませんが、それでもなお、遅延の概念を Java で利用することはできます。

Java での遅延イテレーター

Java には遅延コレクションに対するネイティブなサポートが欠けているとは言え、`Iterator` を使えば、遅延コレクションをシミュレートすることができます。連載のこれまでの記事と同様に、今回も単純な素数アルゴリズムを使用して関数型での概念を説明します。前回の記事で最適化したクラスをベースとして、リスト 1 に示す機能強化を行います。

リスト 1. 素数を判別する単純なアルゴリズム

```
import java.util.HashSet;
import java.util.Set;
import static java.lang.Math.sqrt;

public class Prime {

    public static boolean isFactor(int potential, int number) {
        return number % potential == 0;
    }

    public static Set<Integer> getFactors(int number) {
        Set<Integer> factors = new HashSet<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < sqrt(number) + 1; i++)
            if (isFactor(i, number)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    public static int sumFactors(int number) {
        int sum = 0;
        for (int i : getFactors(number))
            sum += i;
        return sum;
    }

    public static boolean isPrime(int number) {
        return number == 2 || sumFactors(number) == number + 1;
    }

    public static Integer nextPrimeFrom(int lastPrime) {
        lastPrime++;
        while (! isPrime(lastPrime)) lastPrime++;
        return lastPrime;
```

```
    }
}
```

ある整数が素数であるかどうかをこのクラスでどのように判別するかについては、前々回の記事でクラス内部の詳細を説明しました。リスト 1 で私が追加したのは、渡された引数をベースに次の素数を生成する `nextPrimeFrom()` メソッドです。このメソッドが、以降に記載する例で活躍します。

一般に開発者は、バッキング・ストアとしてコレクションを使用することをイテレーターとみなしますが、`Iterator` インターフェースをサポートするものはすべてイテレーターであると言えます。したがって、リスト 2 に記載するような、素数の無限イテレーターを作成することができます。

リスト 2. 遅延イテレーターを作成する

```
public class PrimeIterator implements Iterator<Integer> {
    private int lastPrime = 1;

    public boolean hasNext() {
        return true;
    }

    public Integer next() {
        return lastPrime = Prime.nextPrimeFrom(lastPrime);
    }

    public void remove() {
       throw new RuntimeException("Can't change the fundamental nature of the universe!");
    }
}
```

リスト 2 の `hasNext()` メソッドは、常に `true` を返します。私たちが知る限りでは、素数の数は無限だからです。`remove()` メソッドはここでは該当しないので、誤って呼び出された場合には例外をスローします。ここで大きく貢献するメソッドは `next()` メソッドです。このメソッドはたった 1 行で 2 つの仕事をこなします。その 1 つは、リスト 1 で追加した`nextPrimeFrom()` メソッドを呼び出すことによって、最後の素数をベースに次の素数を生成することです。もう 1 つは、単一のステートメントで値を代入して返すことができる Java の能力を活用して、内部 `lastPrime` フィールドを更新することです。リスト 3 で、この遅延イテレーターを使ってみます。

リスト 3. 遅延イテレーターをテストする

```
public class PrimeTest {
    private ArrayList<Integer> PRIMES_BELOW_50 = new ArrayList<Integer>() {{
                add(2);  add(3);  add(5);  add(7);  add(11);  add(13);
                add(17); add(19); add(23); add(29); add(31);  add(37);
                add(41); add(43); add(47);
            }};

    @Test
    public void prime_iterator() {
        Iterator<Integer> it = new PrimeIterator();
        for (int i : PRIMES_BELOW_50) {
            assertTrue(i == it.next());
        }
    }
}
```

リスト 3 では、PrimeIterator を作成し、このイテレーターが最初の 50 個の素数を報告することを検証します。これはイテレーターの一般的な使い方ではありませんが、遅延コレクションの有用な振る舞いの一部を模倣していることは確かです。

LazyList を使用する

Jakarta Commons に含まれている LazyList クラス (「参考文献」を参照) は、Decorator デザイン・パターンとファクトリーを組み合わせて使用します。Commons の LazyList を使用するには、既存のリストをラップして遅延リストに変え、新しい値を生成するファクトリーを作成する必要があります。リスト 4 に記載する LazyList の使用例を見てください。

リスト 4. Commons の LazyList をテストする

```
public class PrimeTest {
    private ArrayList<Integer> PRIMES_BELOW_50 = new ArrayList<Integer>() {{
                add(2);  add(3);  add(5);  add(7);  add(11);  add(13);
                add(17); add(19); add(23); add(29); add(31);  add(37);
                add(41); add(43); add(47);
            }};

    @Test
    public void prime_factory() {
        List<Integer> primes = new ArrayList<Integer>();
        List<Integer> lazyPrimes = LazyList.decorate(primes, new PrimeFactory());
        for (int i = 0; i < PRIMES_BELOW_50.size(); i++)
            assertEquals(PRIMES_BELOW_50.get(i), lazyPrimes.get(i));
    }
}
```

リスト 4 では、新しい空の ArrayList を作成し、新しい値を生成するための PrimeFactory と併せて Commons の LazyList.decorate() メソッド内にラップします。Commons の LazyList はリストにすでに含まれているあらゆる値を使用しますが、まだ値のないインデックスに対して get() メソッドが呼び出されると、LazyList はファクトリー (この例では PrimeFactory()) を使用して値を生成して取り込みます。リスト 5 に PrimeFactory を記載します。

リスト 5. LazyList で使用される PrimeFactory

```
public class PrimeFactory implements Factory {
    private int index = 0;

    @Override
    public Object create() {
        return Prime.indexedPrime(index++);
    }
}
```

あらゆる遅延リストには、後続の値を生成する方法が必要です。その方法として、リスト 2 では next() メソッドと Prime の nextPrimeFrom() メソッドの組み合わせを使用しました。リスト 4 の Commons の LazyLists では、PrimeFactory インスタンスを使用しています。

Commons の LazyList 実装の特異な点は、新しい値の要求時にファクトリー・メソッドに渡される情報が少ないことです。設計からして、要求された要素のインデックスを渡すようにすらなっていないので、現行の状態を維持する役目は、PrimeFactory クラスに押し付けられます。このこ

とから、バッキング・リストに対する望ましくない依存関係がもたらされます (インデックス番号を `PrimeFactory` の内部状態と同期させるために、空のリストとしてリストを初期化する必要があるためです)。Commons の `LazyList` はひいき目にみても、初歩的な実装というところがせいぜいです。それよりも遥かに優れたオープンソースの実装があります。その 1 つは、Totally Lazy です。

Totally Lazy

Totally Lazy は、Java にファーストクラスの遅延を追加するフレームワークです (「参考文献」を参照)。Totally Lazy については前々回の記事で取り上げましたが、そのイディオムについては取り上げませんでした。このフレームワークの目標の 1 つは、静的インポートの組み合わせを使用して、極めて読みやすいコードを作成することです。リスト 6 に、この Totally Lazy の機能をフル活用して作成した単純な素数ファインダーを記載します。

リスト 6. Totally Lazy による静的インポートのフル活用

```
import com.googlecode.totallylazy.Predicate;
import com.googlecode.totallylazy.Sequence;

import static com.googlecode.totallylazy.Predicates.is;
import static com.googlecode.totallylazy.numbers.Numbers.equalTo;
import static com.googlecode.totallylazy.numbers.Numbers.increment;
import static com.googlecode.totallylazy.numbers.Numbers.range;
import static com.googlecode.totallylazy.numbers.Numbers.remainder;
import static com.googlecode.totallylazy.numbers.Numbers.sum;
import static com.googlecode.totallylazy.numbers.Numbers.zero;
import static com.googlecode.totallylazy.predicates.WherePredicate.where;

public class Prime {
    public static Predicate<Number> isFactor(Number n) {
        return where(remainder(n), is(zero));
    }

    public static Sequence<Number> factors(Number n){
        return range(1, n).filter(isFactor(n));
    }

    public static Number sumFactors(Number n){
        return factors(n).reduce(sum);
    }

    public static boolean isPrime(Number n){
        return equalTo(increment(n), sumFactors(n));
    }
}
```

リスト 6 では、静的インポートが完了した後のコードは、代表的な Java コードではないものの、非常に読みやすくなっています。Totally Lazy は一部、JUnit 対応 Hamcrest テスト拡張機能の流れるようなインターフェース (「参考文献」を参照) に発想を得ていて、Hamcrest のクラスをいくつか使用します。上記では、`isFactor()` メソッドが、Totally Lazy の `remainder()` メソッドと Hamcrest の `is()` メソッドを併せて引数に指定した `where()` メソッドの呼び出しとなっています。同様に、`factors()` メソッドは `range()` オブジェクトの `filter()` メソッドの呼び出しとなり、今ではすっかりお馴染みの `reduce()` メソッドを使用して合計を判断します。最後に、`isPrime()` メソッドが Hamcrest の `equalTo()` メソッドを使用して、約数の合計とインクリメントされた値とが等しいかどうかを判別します。

洞察力のある読者は、約数を判別するためにもっと効率的なアルゴリズムを使用すれば、前回の記事で説明した最適化をリスト 6 の実装で実装できることにお気付きでしょう。そのように最適化したバージョンをリスト 7 に記載します。

リスト 7. 最適化された素数検出プログラムの Totally Lazy 実装

```
public class PrimeFast {
    public static Predicate<Number> isFactor(Number n) {
        return where(remainder(n), is(zero));
    }

    public static Sequence<Number> getFactors(final Number n){
        Sequence<Number> lowerRange = range(1, squareRoot(n)).filter(isFactor(n));
        return lowerRange.join(lowerRange.map(divide().apply(n)));
    }

    public static Sequence<Number> factors(final Number n) {
        return getFactors(n).memorise();
    }

    public static Number sumFactors(Number n){
        return factors(n).reduce(sum);
    }

    public static boolean isPrime(Number n){
        return equalTo(increment(n), sumFactors(n));
    }
}
```

リスト 7 に加えられている主な変更は 2 つです。まず、`getFactors()` アルゴリズムを改善して、平方根までの約数を取得した後に、平方根より大きい約数のペアを生成するようにしました。Totally Lazy では、`divide()` のような処理でさえも、流れるようなインターフェース・スタイルで表現することができます。2 つ目の変更には、メモ化が伴います。メモ化とは、同じ引数を持つ関数呼び出しを自動的にキャッシュする手法です。上記では、`getFactors()` メソッドをメモ化した `factors()` メソッドを使用するように `sumFactors()` メソッドを変更してあります。Totally Lazy はこのフレームワークの一部としてメモ化を実装するため、この最適化を実装するために必要となる追加コードはありません。ただし、このフレームワークの作成者は、より一般的に (Groovy で) 使用されている `memoize()` ではなく、`memorise()` という綴りのメソッド名にしていることに注意してください。

その名のとおり、Totally Lazy はフレームワーク全体にわたって可能な限り遅延を使用しようとします。実際、Totally Lazy フレームワーク自体に組み込まれている `primes()` ジェネレーターは、このフレームワークのビルディング・ブロックを使用して素数の無限数列を実装します。リスト 8 に記載する `Numbers` クラスの抜粋を見てください。

リスト 8. 無限の素数を実装する Totally Lazy の抜粋

```
public static Function1<Number, Number> nextPrime = new Function1<Number, Number>() {
       @Override
       public Number call(Number number) throws Exception {
              return nextPrime(number);
      }
};

public static Computation<Number> primes = computation(2, computation(3, nextPrime));
```

```
public static Sequence<Number> primes() {
       return primes;
}

public static LogicalPredicate<Number> prime = new LogicalPredicate<Number>() {
    public final boolean matches(final Number candidate) {
        return isPrime(candidate);
    }

};

public static Number nextPrime(Number number) {
    return iterate(add(2), number).filter(prime).second();
}
```

`nextPrime()` メソッドは `Function1` を新規に作成します。`Function1` は、Totally Lazy による擬似高階関数の実装であり、単一の引数 `Number` を取って、結果として `Number` を生成するように作られています。この例での `Function1` は、`nextPrime()` メソッドの結果を返します。素数の状態を保持するために作成された `primes` 変数は、(最初の素数である) `2` をシード値とする computation を実行し、その中で次の素数を求めるための computation を使用しています。これは、遅延実装の典型的なパターンです。つまり、次の要素に加え、以降の値を生成するためのメソッドを持っています。

リスト 8 の `nextPrime()` メソッドの結果を判別するために、Totally Lazy は新しい `LogicalPredicate` を作成して、そこに素数であるかどうかの判別をカプセル化します。続いて作成する `nextPrime()` メソッドでは、Totally Lazy に含まれる、流れるようなインターフェースを使用して次の素数を判別します。

Totally Lazy は、コードを極めて読みやすくすることに関して、Java で静的インポートを使用するという方法で素晴らしい成果を上げます。開発者の多くは、内部のドメイン特化言語のホストとしては、Java は物足りないと考えていますが、Totally Lazy はその考えを覆します。Totally Lazy は積極的に遅延を使用して、可能な限りあらゆる操作を遅延させます。

まとめ

今回の記事では、最初にイテレーターを使用して Java で遅延コレクションをシミュレートして作成した後、Jakarta Commons Collections の初歩的な `LazyList` クラスを使用することで、遅延について探りました。そして最後に Totally Lazy で、素数であるかどうかを内部で判別する遅延コレクションと、素数の遅延無限コレクションを両方とも使用したサンプル・コードを実装しました。また、コードの読みやすさを改善するための静的インポートを使用して、Totally Lazy の流れるようなインターフェース・スタイルの表現力も明らかにしました。

次回の記事でも引き続き遅延処理について探り、Groovy、Scala、Clojure での遅延処理を取り上げます。

著者について

Neal Ford

Neal Ford は世界的な IT コンサルティング企業である ThoughtWorks のソフトウェア・アーキテクトであり、Meme Wrangler でもあります。また彼は、アプリケーション、教育資料、雑誌記事、コースウェア、ビデオや DVD によるプレゼンテーションなどの設計と開発も行っています。さまざまな技術に関する本の著者、編集者でもあり、最新の著書は『プロダクティブ・プログラマ — プログラマのための生産性向上術』です。彼は大規模なエンタープライズ・アプリケーションの設計や構築を専門にしています。また彼は世界各地で開催される開発者会議での講演者としても国際的に有名です。彼の Web サイトをご覧ください。