今まで知らなかった 5 つの事項: Java 6 コレクション API の場合: 第 2 回

mutable に注意

Ted Neward
Principal
Neward & Associates

2017年 8月 31日
(初版 2010年 5月 04日)

Java のコレクションはどこででも使えますが、軽く考えてはいけません。コレクションには知られていない性質があり、適切に扱わないとトラブルにつながる場合があります。今回の「今まで知らなかった 5 つの事項」では、Ted Neward が Java コレクション API の可変かつ複雑な側面を説明します。このヒントによって、Iterable、HashMap、SortedSet を一層活用できるようになり、コードにバグが発生しなくなるはずです。

このシリーズの他の記事を見る

`java.util` の Collections クラスは、配列を置き換えることで Java のパフォーマンスを改善できるように設計されています。「前回の記事」で学んだように、Collections クラスは柔軟に作られているため、ありとあらゆる方法で Collections クラスをカスタマイズして拡張し、適切で簡潔なコードを生成することができます。

その一方で、コレクションは強力であり、可変でもあります。コレクションの扱いには注意が必要であり、乱用した場合のリスクを認識する必要があります。

1. List は配列と同じではありません

Java 開発者は `ArrayList` が単に Java の配列の置き換えであると思ってしまいがちです。コレクションは配列をベースにしているため、コレクション内の項目をランダムに参照する場合には優れたパフォーマンスを示します。またコレクションは、配列と同様に整数の序数を使って特定の項目を取得します。しかしそうであっても、コレクションで配列を直接置き換えることはできません。

コレクションと配列を区別するためには、『順序』と『位置』の違いを認識する必要があります。例えば、`List` は順序を保持するインターフェースであり、コレクションに配置された項目の順序が保持されます (リスト 1)

リスト 1. 可変型のキー

```
import java.util.*;

public class OrderAndPosition
{
    public static <T> void dumpArray(T[] array)
    {
        System.out.println("=============");
        for (int i=0; i<array.length; i++)
            System.out.println("Position " + i + ": " + array[i]);
    }
    public static <T> void dumpList(List<T> list)
    {
        System.out.println("=============");
        for (int i=0; i<list.size(); i++)
            System.out.println("Ordinal " + i + ": " + list.get(i));
    }

    public static void main(String[] args)
    {
        List<String> argList = new ArrayList<String>(Arrays.asList(args));

        dumpArray(args);
        args[1] = null;
        dumpArray(args);

        dumpList(argList);
        argList.remove(1);
        dumpList(argList);
    }
}
```

上記の `List` から 3 番目の要素が削除されると、その要素の「後ろ」にある他の要素は前に移動し、空いた場所が埋まります。明らかに、このコレクションの動作は配列の動作とは異なります。(実際、配列から項目を削除すること自体、List から項目を削除することと完全に同じではありません。配列から項目を「削除する」ということは、その項目のインデックスの場所を新しい参照またはヌルで上書きすることを意味します。)

2. Iterator に驚いてはいけません

Java 開発者が Java Collections の `Iterator` を好むことは確かです。しかし皆さんが `Iterator` インターフェースを真剣に調べたのはいつのことでしょう。言ってみれば、私達はほとんどの場合、`Iterator` を `for()` ループや拡張 `for()` ループの中に配置するだけで、すぐにその先へ進んでしまいがちです。

しかし詳しく調べると、`Iterator` には驚くべき 2 つの側面があります。

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Ted Neward が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、ほとんど知られていない事実を紹介します。

第 1 に、`Iterator` はソース・コレクションから安全にオブジェクトを削除する機能を持っており、そのためには `Iterator` 自体の `remove()` を呼び出せばよいのです。この場合のポイントは、`ConcurrentModifiedException` の発生を避けることです。`ConcurrentModifiedException` は名

前どおりの内容を通知します。つまり、`Iterator` によってコレクションを操作している最中にそのコレクションが変更されたことを通知します。一部のコレクションでは、`Collection` に対する繰り返しの処理を実行中であっても要素の追加や削除を行えますが、`Iterator` の `remove()` を呼び出した方がより安全なプラクティスになります。

第 2 に、`Iterator` は、`Iterator` から派生した (おそらくもっと強力な) 仲間をサポートしています。`ListIterator` は `List` からでないと利用できませんが、繰り返しの処理を実行中の `List` に対する追加と削除の両方をサポートしており、また `List` を上下にスクロールすることができます。

上下スクロールは、どこにでもあるような「結果セットをスクロールする」シナリオで、特に威力を発揮します。つまりデータベースやその他のコレクションから取得した多くの結果から 10 件のみを表示するような場合です。また上下スクロールは、コレクションやリストに対する繰り返しの処理を、先頭からではなく末尾から行う場合にも使うことができます。`ListIterator` を使った方が、`List.get()` の整数パラメーターをデクリメントしながら `List` を末尾から処理するよりも、はるかに簡単です。

3. すべての Iterable がコレクションに由来するとは限りません

Ruby 開発者や Groovy 開発者は、1 行のコードで、いかにしてテキスト・ファイル全体にわたる繰り返し処理を行い、その内容を出力できるかを自慢したがります。彼らによれば、ほとんどの場合、これと同じことを Java プログラムで行おうとすると何十行ものコードが必要になります。つまり `FileReader` を開いてから `BufferedReader` を開き、続いて `while()` ループを作成して `getLine()` を呼び出し、ヌルが返されるまでそれを続けます。そしてもちろん、これらのすべてを `try/catch/finally` ブロックの中で行う必要があり、このブロックが例外を処理し、終了時にはファイル・ハンドルを閉じる必要があります。

これは下らない自慢話に思えますが、このようなコードを作成することにはメリットがあります。

彼ら (そして非常に多くの Java 開発者) は知らないかもしれませんが、すべての `Iterable` がコレクションから生まれるわけではありません。実は `Iterable` は `Iterator` を作成することができます。この `Iterator` は次の要素をどこからか作り出す方法を知っており、盲目的に既存の `Collection` から次の要素を取得するわけではありません。

リスト 2. ファイルに対する繰り返し処理

```
// FileUtils.java
import java.io.*;
import java.util.*;

public class FileUtils
{
    public static Iterable<String> readlines(String filename)
     throws IOException
    {
     final FileReader fr = new FileReader(filename);
     final BufferedReader br = new BufferedReader(fr);

     return new Iterable<String>() {
      public <code>Iterator</code><String> iterator() {
       return new <code>Iterator</code><String>() {
        public boolean hasNext() {
         return line != null;
```

```
        }
        public String next() {
         String retval = line;
         line = getLine();
         return retval;
        }
        public void remove() {
         throw new UnsupportedOperationException();
        }
        String getLine() {
         String line = null;
         try {
          line = br.readLine();
         }
         catch (IOException ioEx) {
          line = null;
         }
         return line;
        }
        String line = getLine();
       };
      }
     };
    }
}

//DumpApp.java
import java.util.*;

public class DumpApp
{
    public static void main(String[] args)
        throws Exception
    {
        for (String line : FileUtils.readlines(args[0]))
            System.out.println(line);
    }
}
```

この方法には、ファイルの内容すべてをメモリーに保持する必要がないというメリットがあります。しかしコードを見るとわかるように、この方法ではベースとなるファイル・ハンドルに対して `close()` が実行されていないことに注意する必要があります。(この問題は、`readLine()` がヌルを返した時点で必ずクローズするように修正できますが、`Iterator` が最後まで実行されないケースを解決することはできません。)

4. 可変の `hashCode()` に注意

`Map` は素晴らしいコレクションであり、Perl などの他の言語によく見られる、キーと値のペアのコレクションのスマートさを実現してくれます。また JDK には HashMap という形で優れた `Map` 実装が用意されており、`HashMap` は内部でハッシュテーブルを使ってキーに対応する値を素早く検索します。しかしここで少し問題があります。可変フィールドの内容に依存するハッシュ・コードをサポートするキーはバグの影響を受けやすいため、どんなに忍耐強い Java 開発者でも頭が変になってしまうことでしょう。

リスト 3 の `Person` オブジェクトに通常の `hashCode()` があるとすると (Person オブジェクトは `firstName` フィールド、`lastName` フィールド、`age` フィールド (どれも final ではありません) を使って `hashCode()` を計算します)、`Map` に対して `get()` を呼び出すと失敗し、`null` が返されます。

リスト 3. 可変の `hashCode()` によってバグが発生しやすくなる

```
// Person.java
import java.util.*;

public class Person
    implements Iterable<Person>
{
    public Person(String fn, String ln, int a, Person... kids)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
        for (Person kid : kids)
            children.add(kid);
    }

    // ...

    public void setFirstName(String value) { this.firstName = value; }
    public void setLastName(String value) { this.lastName = value; }
    public void setAge(int value) { this.age = value; }

    public int hashCode() {
        return firstName.hashCode() & lastName.hashCode() & age;
    }

    // ...

    private String firstName;
    private String lastName;
    private int age;
    private List<Person> children = new ArrayList<Person>();
}


// MissingHash.java
import java.util.*;

public class MissingHash
{
    public static void main(String[] args)
    {
        Person p1 = new Person("Ted", "Neward", 39);
        Person p2 = new Person("Charlotte", "Neward", 38);
        System.out.println(p1.hashCode());

        Map<Person, Person> map = new HashMap<Person, Person>();
        map.put(p1, p2);

        p1.setLastName("Finkelstein");
        System.out.println(p1.hashCode());

        System.out.println(map.get(p1));
    }
}
```

明らかに、この方法には問題がありますが、解決方法は簡単です。`HashMap` のキーとして可変オブジェクト型を使わなければよいのです。

5. `equals()` と `Comparable`

Java 開発者が Javadoc を調べていると、頻繁に `SortedSet` 型 (そして JDK の中で唯一それを実装した `TreeSet`) に遭遇します。`SortedSet` は `java.util` パッケージの中で唯一ソート動作を提供する

Collection なので、開発者は詳細をあまり調べずに SortedSet を使い始めてしまいがちです。それを示したものがリスト 4 です。

リスト 4. SortedSet に注意

```
import java.util.*;

public class UsingSortedSet
{
    public static void main(String[] args)
    {
        List<Person> persons = Arrays.asList(
            new Person("Ted", "Neward", 39),
            new Person("Ron", "Reynolds", 39),
            new Person("Charlotte", "Neward", 38),
            new Person("Matthew", "McCullough", 18)
        );
        SortedSet ss = new TreeSet(new Comparator<Person>() {
            public int compare(Person lhs, Person rhs) {
                return lhs.getLastName().compareTo(rhs.getLastName());
            }
        });
        ss.addAll(perons);
        System.out.println(ss);
    }
}
```

このコードをしばらく使っていると、Set のコア機能の 1 つ、つまり重複が許されないということに気が付きます。この機能については、実際に Set の Javadoc に記述されています。それによれば、Set は「重複要素を含まないコレクションです。もっと厳密に言えば、Set は e1.equals(e2) となるような e1 要素と e2 要素のペアを含まず、またヌル要素は最大で 1 つしか含みません。」

しかし、リスト 4 が実際にこの説明に当てはまるようには思えません。リスト 4 の Person オブジェクトは (Person に equals() を実行してみると) どれもそれぞれに異なりますが、出力してみると、TreeSet の中には 3 つのオブジェクトしかありません。

Set の性質は上記のように記述されていますが、TreeSet では、オブジェクトが直接 Comparable を実装するか、あるいはオブジェクトを作成する際にオブジェクトに Comparator を渡す必要があり、equals() を使ってオブジェクトを比較するのではなく、Comparator/Comparable の compare メソッドまたは compareTo メソッドを使います。

つまり、Set に保存されたオブジェクト同士が等しいと判断するための方法としては、Javadoc で想定しているとおりの equals() メソッドを使う方法と、Comparable/Comparator メソッドを使う方法の 2 通りの方法があり、どちらを使うかは人によって異なるのです。

もっと悪いことに、2 つのオブジェクトが同じものであると宣言するだけでは十分ではありません。なぜなら、ソートのための比較は、等価かどうかを判断するための比較と同じではないからです。例えば、苗字でソートする際に 2 つの Person が等価であると見なしてもまったく問題ないかもしれませんが、内容を比較する場合には、その 2 つの Person が等価であると見なすことはできないかもしれません。

Set を実装する際に、equals() と、Comparable.compareTo() が 0 を返すこと、との間に違いがある場合は、その違いを必ず明確にして、ドキュメントに明記する必要があります。

まとめ

Java Collections ライブラリーには、知ってさえいれば作業をはるかに楽にでき、生産性を高められるちょっとしたツールが散在しています。しかしそうしたちょっとしたツールを見つけるには、少し複雑な作業が必要になるケースがよくあります。例えば、`HashMap` を使って構いませんが、キーとして絶対に可変オブジェクト型を使ってはならない、といった具合です。

ダウンロード

内容	ファイル名	サイズ
Source code for the places application	j-5things3-src.zip	15KB

著者について

Ted Neward

Ted Neward has written over 250 articles and a dozen books across many different technologies, including .NET, iOS, Java, Android, and JavaScript. He resides in Seattle with his wife, two kids, nine laptops, fourteen mobile devices, and two cats. Email him if you're interested in having him or his company work with you.