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 1 回

Java コレクションをカスタマイズし、拡張する

Ted Neward
Principal
Neward & Associates

2017年 8月 31日
(初版 2010年 4月 20日)

Java 6 コレクション API は単なる配列の置き換え以上の機能を持っていますが、配列の置き換えは出発点として悪くはありません。この記事では Ted Neward が、Java コレクション API のカスタマイズ方法や拡張方法の基本を含め、コレクションをさらに活用するための 5 つのヒントを紹介します。

このシリーズの他の記事を見る

Java コレクション API は多くの Java 開発者にとって、標準の Java 配列および Java 配列の多くの欠点を置き換えるものとして、強く望まれていたものでした。コレクションと `ArrayList` とを基本的に関連付けることは誤りではありませんが、追求してみると、コレクションにはもっと多くの機能が含まれているのです。

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。

同様に、`Map` (そしてその実装としてよく選ばれる `HashMap`) は、名前と値のペア、またはキーと値のペアを扱う際に非常に便利なものですが、そうした用途のみに Map を限定する理由は何もありません。適切な API を使うことで、さらには適切なコレクションを使うことで、エラーを起こしやすいコードを大量に修正することができます。

この連載記事「今まで知らなかった 5 つの事項」の第 2 回目の今回は、コレクションについて説明する数回からなる記事の第 1 回目です。このようにしてコレクションを取り上げる理由は、コレクションは Java プログラミングの中で非常に中心的なものであるからです。ここではまず、例えば `Array` を `List` で置き換えるといった日常的なことを、最も手軽に行う方法について調べます (ただしこの方法は、最も一般的な方法とは言えないかもしれません)。次に、あまり知られていな

い内容、例えばカスタムのコレクション・クラスの作成方法や Java コレクション API の拡張方法などについて調べます。

1. コレクションは配列よりも優れている

Java 技術を初めて学ぶ開発者は知らないかもしれませんが、配列が Java 言語に含まれることになったそもそもの理由は、1990年代の初め頃に C++ 開発者達から挙げられた、Java のパフォーマンスに対する批判をかわすためでした。それ以来長い年月が経ち、配列のパフォーマンスは一般に Java コレクション・ライブラリーのパフォーマンスと比べて見劣りするものになっています。

例えば配列の内容をストリングにするためには、配列に対して繰り返し処理を行い、配列の内容同士を連結して `String` にする必要があります。一方コレクションの実装にはすべて、有効な `toString()` の実装があります。

適切なプラクティスとしては、ごく一部の例外を除き、配列を見たら可能な限り迅速にコレクションに変換することです。では、そのような変換を最も簡単に行うにはどのような方法を使うとよいのでしょう？実は、Java コレクション API を使うことで、そうした変換を容易に行えるのです (リスト 1)。

リスト 1. ArrayToList

```
import java.util.*;

public class ArrayToList
{
    public static void main(String[] args)
    {
        // This gives us nothing good
        System.out.println(args);

        // Convert args to a List of String
        List<String> argList = Arrays.asList(args);

        // Print them out
        System.out.println(argList);
    }
}
```

返される `List` は変更できないことに注意してください。そのため、この List に新しい要素を追加しようとすると、`UnsupportedOperationException` がスローされます。

また、`Arrays.asList()` は varargs パラメーターを使って要素を `List` に追加するため、Arrays.asList() を使うことで、`new` によって作成されたオブジェクトから容易に `List` を作成することもできます。

2. 繰り返しは非効率

あるコレクション (特に、配列から作成されたコレクション) の内容を別のコレクションに移動したい、または大きなオブジェクト・コレクションから小さなオブジェクト・コレクションを削除したい、といった要求は特別なものではありません。

そうした場合、単純にコレクションに対して繰り返し処理を実行し、要素が見つかるごとにその要素を追加または削除したくなるかもしれません。しかしそんなことをしてはなりません。

この場合、繰り返し処理には以下のように大きな欠点があります。

- 追加や削除を行うごとにコレクションをリサイズするのは非効率です。
- ロックの取得、操作の実行、そしてロックの解放を毎回行っていると、並行処理の悪夢が起こる可能性があります。
- 追加や削除が行われている間、他のスレッドがそのコレクションを操作しようとして競合状態が発生します。

こうした問題はすべて、追加または削除の対象要素を含むコレクションを `addAll` または `removeAll` を使って渡すことで回避することができます。

3. すべての Iterable に for ループを使う

強化された for ループは Java 5 で Java 言語に追加された非常に便利な機能の 1 つです。この強化された for ループによって、Java のコレクションを処理する上での最後の障害が取り除かれたのです。

Java 5 以前では、手動で `Iterator` を取得し、その `Iterator` によって指定されたオブジェクトを `next()` を使って取得し、他にもオブジェクトがあるかどうかを `hasNext()` を使って調べる必要がありました。Java 5 以降では、気にせず for ループ変数を使うことで、上記のすべてが暗黙のうちに処理されます。

実は、この強化された for ループは `Iterable` インターフェースを実装するすべてのオブジェクトに有効であり、`Collections` に対してのみ有効なわけではありません。

リスト 2 は、`Iterator` として得られる `Person` オブジェクトから children のリストを作成するための1 つの方法を示しています。内部の `List` に対する参照を渡す (そうすると、`Person` の外の呼び出し側が皆さんの家族に子供を追加できてしまいます。これは大半の親にとって望ましくない動作です) 代わりに、`Person` 型が `Iterable` を実装しています。またこの方法では、強化された for ループによって children をウォークスルーすることもできます。

リスト 2. 強化された for ループ: children を紹介する

```
// 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 child : kids)
            children.add(child);
    }
    public String getFirstName() { return this.firstName; }
    public String getLastName() { return this.lastName; }
    public int getAge() { return this.age; }

    public Iterator<Person> iterator() { return children.iterator(); }

    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 String toString() {
        return "[Person: " +
            "firstName=" + firstName + " " +
            "lastName=" + lastName + " " +
            "age=" + age + "]";
    }

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

// App.java
public class App
{
    public static void main(String[] args)
    {
        Person ted = new Person("Ted", "Neward", 39,
            new Person("Michael", "Neward", 16),
            new Person("Matthew", "Neward", 10));

        // Iterate over the kids
        for (Person kid : ted)
        {
            System.out.println(kid.getFirstName());
        }
    }
}
```

`Iterable` を使用すると、ドメインのモデリングの際、いくつか明確な問題が生じます。それは、`iterator()` メソッドによって「暗黙的に」サポートできるオブジェクトのコレクションは 1 つのみだからです。しかし child コレクションが当然で明確な場合には、`Iterable` を使うことでドメイン型に対するプログラミングが非常に容易になり、またわかりやすくなります。

4. 昔ながらのアルゴリズムとカスタム・アルゴリズム

皆さんは `Collection` を逆にウォークスルーしたいと思ったことがありませんか。そうした場合には昔ながらの Java コレクション・アルゴリズムが便利なのです。

上記のリスト 2 の `Person` の children は、渡された順に並んでいます。しかし今度は、逆の順序で並べたいとします。別の for ループを作成して各オブジェクトを新しい `ArrayList` に逆の順序で挿入することもできますが、それが 3 つ、4 つとなるとコーディングが面倒になってきます。

そうした場合に、あまり活用されていないアルゴリズム (リスト 3) を使うことができます。

リスト 3. ReverseIterator

```
public class ReverseIterator
{
    public static void main(String[] args)
    {
        Person ted = new Person("Ted", "Neward", 39,
            new Person("Michael", "Neward", 16),
            new Person("Matthew", "Neward", 10));

        // Make a copy of the List
        List<Person> kids = new ArrayList<Person>(ted.getChildren());
        // Reverse it
        Collections.reverse(kids);
        // Display it
        System.out.println(kids);
    }
}
```

`Collections` クラスには、こうした「アルゴリズム」がいくつかあります。これらの「アルゴリズム」は静的メソッドであり、`Collections` を引数に取るように実装され、実装とは独立してコレクション全体を動作させることができます。

しかも、この優れた API の設計は、`Collections` クラス上に存在するアルゴリズムのみで終わりではありません。例えば私の好きなメソッドとして、(渡されるコレクションの) 内容を直接変更しないメソッドも可能です。つまり便利なことに、カスタムのアルゴリズムを独自に作成できるのです。その一例がリスト 4 です。

リスト 4. 単純化した ReverseIterator

```
class MyCollections
{
    public static <T> List<T> reverse(List<T> src)
    {
        List<T> results = new ArrayList<T>(src);
        Collections.reverse(results);
        return results;
    }
}
```

5. コレクション API を拡張する

上記のカスタマイズされたアルゴリズムは、この記事で説明する Java コレクション API についての最後のポイントを示しています。つまり Java コレクション API は昔から、開発者の特定の目的に合わせて拡張、加工できるように作られていたのです。

そのため、例えば `Person` クラスの children のリストを必ず年齢でソートする必要があったとしましょう。children のソートを何度も繰り返すコードを (例えば `Collections.sort` メソッドを使って) 作成することもできますが、ソートを実行してくれる `Collection` クラスがあれば、その方がはるかに望ましいはずです。

実際、`Collection` へのオブジェクトの挿入順を保持することすら気にする必要はないかもしれません (`List` を使う主な理由は順序を気にするからです)。しかしソートされた順序でオブジェクトを保持したい場合もあるかもしれません。

`java.util` の中にある `Collection` クラスのどれを使っても、そうした要件を満たすことはできませんが、そうしたクラスの作成は非常に簡単です。必要なことは、`Collection` が行うべき抽象動作を記述するインターフェースを作成することだけです。`SortedCollection` の場合、目的はその動作そのものです。

リスト 5. SortedCollection

```
public interface SortedCollection<E> extends Collection<E>
{
    public Comparator<E> getComparator();
    public void setComparator(Comparator<E> comp);
}
```

この新しいインターフェースの実装の作成は拍子抜けするほど簡単です。

リスト 6. ArraySortedCollection

```
import java.util.*;

public class ArraySortedCollection<E>
    implements SortedCollection<E>, Iterable<E>
{
    private Comparator<E> comparator;
    private ArrayList<E> list;

    public ArraySortedCollection(Comparator<E> c)
    {
        this.list = new ArrayList<E>();
        this.comparator = c;
    }
    public ArraySortedCollection(Collection<? extends E> src, Comparator<E> c)
    {
        this.list = new ArrayList<E>(src);
        this.comparator = c;
        sortThis();
    }

    public Comparator<E> getComparator() { return comparator; }
    public void setComparator(Comparator<E> cmp) { comparator = cmp; sortThis(); }

    public boolean add(E e)
    { boolean r = list.add(e); sortThis(); return r; }
    public boolean addAll(Collection<? extends E> ec)
    { boolean r = list.addAll(ec); sortThis(); return r; }
    public boolean remove(Object o)
    { boolean r = list.remove(o); sortThis(); return r; }
    public boolean removeAll(Collection<?> c)
    { boolean r = list.removeAll(c); sortThis(); return r; }
    public boolean retainAll(Collection<?> ec)
    { boolean r = list.retainAll(ec); sortThis(); return r; }

    public void clear() { list.clear(); }
    public boolean contains(Object o) { return list.contains(o); }
    public boolean containsAll(Collection <?> c) { return list.containsAll(c); }
    public boolean isEmpty() { return list.isEmpty(); }
    public Iterator<E> iterator() { return list.iterator(); }
    public int size() { return list.size(); }
    public Object[] toArray() { return list.toArray(); }
    public <T> T[] toArray(T[] a) { return list.toArray(a); }

    public boolean equals(Object o)
    {
```

```
        if (o == this)
            return true;

        if (o instanceof ArraySortedCollection)
        {
            ArraySortedCollection<E> rhs = (ArraySortedCollection<E>)o;
            return this.list.equals(rhs.list);
        }

        return false;
    }
    public int hashCode()
    {
        return list.hashCode();
    }
    public String toString()
    {
        return list.toString();
    }

    private void sortThis()
    {
        Collections.sort(list, comparator);
    }
}
```

この、にわか仕立ての実装は、最適化をまったく考慮せずに作成してあります。そのため、当然ながら何らかのリファクタリングが必要です。しかしポイントは、Java コレクション API が決してコレクション関連のすべてに対する最終仕様を意図したものではない、ということです。Java コレクション API は拡張が必要であり、また拡張を推奨しているのです。

確かに、`java.util.concurrent` に導入された拡張など、いくつかの拡張は極めて大変な作業を伴うものです。しかしそれ以外は、既存の `Collection` クラスに対してカスタムのアルゴリズムを作成したり、あるいは既存の `Collection` クラスを単純に継承したりすればよいだけです。

Java コレクション API の拡張は非常に面倒と思えるかもしれませんが、実際に作業を始めてみると、決して思ったほど困難ではないことに気づくはずです。

まとめ

Java のシリアライズの場合と同様、Java コレクション API には活用されていない機能がたくさんあります。従って、このテーマを今回の記事だけで終わらせることはできません。次回の「今まで知らなかった 5 つの事項」の記事では、Java 6 コレクション API に関するさらに別の 5 つの事項について説明します。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-5things2-src.zip	10KB

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.