Generics と並行性でコレクションにスパイスを利かせる

Java SE 6 の Java Collections Framework 拡張機能を使用する

John Zukowski (jaz@zukowski.net)
President, JZ Ventures, Inc.
JZ Ventures, Inc.

2008年 4月 08日

Java™ Collections Framework は Java プラットフォームの重要な特徴です。デスクトップ・アプリケーションとエンタープライズ・アプリケーションはいずれも、操作対象の項目を収集するのが一般的だからです。この記事では、Java SE 6 でこのフレームワークに加えられた拡張機能を利用しながらコレクションを操作する方法を紹介します。Generics と並行性を利用することで、`HashMap` や `TreeSet` よりも遥かにアプリケーションの管理および拡張が容易になります。

Java 2 プラットフォームのバージョン 1.2 で初めてリリースされて以来、Java Collections Framework は大きな進展を遂げています。Java SE 5 リリースでは、Generics がこのフレームワークを拡張し、`java.util.concurrent` の導入によって並行性が直接サポートされるようになりました (「参考文献」を参照)。そして Java SE 6 でフレームワークに追加されたのは、さらに改善されたコレクションの双方向アクセスです。この記事では、コレクション・ライブラリーが持つこれらの特徴をすべて紹介し、特によく使用される並行性関連の機能を活用する方法を説明します。

この記事で取り上げるのは Web クローラーで、Web サイトの基本 URL を指定して、何らかの目的で使用できる要素をそのサイトから収集します。まずサイト内のある Web ページから一連のリンクを収集し、それからサイト全体をクロールします。そしてタスクはサブタスクに分割して、それぞれ個別のジョブに独立させて実行します。これらのタスクを実行するなかで、Generics とスレッド・プールについて説明し、実際に使用することになります。説明が複雑にならないよう、タスクはスタンドアロンのクライアント・サイド・アプリケーションとして実装します (この記事の主な目的は Web アプリケーションのデプロイメント方法を説明することではありません。しかし追加の演習として自由に、タスクを起動するための Web アプリケーションを作成してみてください)。

読者の皆さんは、Java プラットフォームのプログラム開発について熟知している必要があります。この記事は、ソケット接続を確立するためのネットワーキング、そしてストリームを読み取るための I/O ライブラリーについて読者が十分な知識を持っていることを前提に作成されています。また、Java SE 6 プラットフォームで開発を行える環境も必要です。Sun Microsystems の JDK 6 Update 5 以降、または IBM の最新 SDK for Java バージョン 6 を使用してください。

Generics についての基本知識

Generics の概念が Java プラットフォームに組み込まれたのは、Java SE 5 リリース (「参考文献」を参照) からです。Generics とは、簡単に言えば、コンパイル時にタイプ・セーフなコレクションが作成できるようにしてくれるものです。初期の Java プラットフォーム・バージョンでは、リスト 1 のようにコレクションを作成して項目を追加していました。

リスト 1. コレクションへの項目追加 - 以前の方法

```
List buttonList = new LinkedList();
buttonList.add(new JButton("One"));
buttonList.add(new JButton("Two"));
buttonList.add(new JButton("Three"));
buttonList.add(new JButton("Four"));
```

コレクションから要素を取得するには、コレクション内のオブジェクトの型を把握する必要がありました。そうでないと、要素をキャストして適切なローカル変数に戻せないからです。

```
JButton first = (JButton)buttonList.get(0);
```

キャストして正しい型に戻す必要はなくても、特定のクラス型で何らかの操作を行う必要があるだろうという前提で要素のキャストは行われていました。このやり方が上手くいくのは、以下のように不正な型のオブジェクトを誤ってコレクションに追加してしまうまでの話です。

```
buttonList.add(new JLabel("Five"));
```

この場合、最後の要素を `JButton` として取得しようとすると、実行時にクラス・キャスト例外が発生します。

```
Line 13: JButton last = (JButton)buttonList.get(4);
>java GetIt
Exception in thread "main" java.lang.ClassCastException:
  javax.swing.JLabel cannot be cast to javax.swing.JButton
        at GetIt.main(GetIt.java:13)
```

`JLabel` をコレクションに追加すること自体は間違っていません。しかし、取得を行うコードはコレクション内のすべての要素は同じ型 (上記の例では `JButton`) であることを前提としているので、コレクションから `JLabel` を取得しようとすると `ClassCastException` が発生してしまいます。さらにこの例外は実行時まで発生しません。十分にテストを行っていなければ、アプリケーションがデプロイされるまで例外が発生しないという事態もあり得ます。

ジェネリック・コレクションの使用

ここから足を踏み入れるのは、Generics の世界です。Generics は、開発サイクルの早い段階でコードの問題を解決できるようにします。コレクションを作成し、そのコレクションに `JButton` オブジェクトを追加するのは通常の処理ですが、例えば `JButton` オブジェクトのコレクションがあった場合に、そこに `JLabel` を追加しようとしたとします。するとコンパイラーが不一致を検出し、コンパイル時に警告を出します。

リスト 2 のプログラムは、ユーザーがジェネリック・コレクション (この例では `List<JButton>`) に誤った型の要素を追加しようとすると、コンパイル時にエラー・メッセージを生成します。

リスト 2. Generics を使用したサンプル・コード (コンパイルすると失敗します)

```
import java.util.*;
import javax.swing.*;

public class GetIt {
  public static void main(String args[]) {
    List<JButton> buttonList = new LinkedList<JButton>();
    buttonList.add(new JButton("One"));
    buttonList.add(new JButton("Two"));
    buttonList.add(new JButton("Three"));
    buttonList.add(new JButton("Four"));
    JButton first = buttonList.get(0);
    buttonList.add(new JLabel("Five"));
    JButton last = buttonList.get(4);
  }
}
```

このプログラムを保存してコンパイルすると、最後に `add()` を呼び出すところでエラーになっていることがわかります。

```
>javac GetIt.java
GetIt.java:12: cannot find symbol
symbol  : method add(javax.swing.JLabel)
location: interface java.util.List<javax.swing.JButton>
                buttonList.add(new JLabel("Five"));
                           ^
1 error
```

上記のエラー・メッセージの 2 行目には、3 行目で `JButton` オブジェクトの `List` としてレポートされている対象に `JLabel` を追加しようとしていることが示されています。この場合、コレクションはどうしても `Component` オブジェクトのコレクションでなければならないのか (Swing プラットフォームのコンポーネントにこだわる場合には、`JComponent` でなければならないのか)、あるいは最初から `JLabel` を追加しようとすべきではなかったのかを判断する必要に迫られます。

リスト 2 で注目すべきことは、コレクションから項目を取得する際に正しい型にキャストしなくてもコレクションから項目を取得できるという点です。コレクションが特定の型として記述されているので、そのコレクションから項目を取得するためのすべての呼び出しは、その特定の型を返します。

このように、Generics を使用するとコード・ベースの管理が遥かに容易になります。この管理の容易さが特に明らかになるのは、コード・ベースが大きくなったためにコード要素を再利用可能なライブラリーに変換する場合です。ライブラリーのユーザーがコレクションに含まれるオブジェクトの型に制約があるのかどうかを心配する必要はもうありません。適切に定義されたメソッドが、その定義によって該当する型を組み込むからです。もしその型から逸脱したとしても、コンパイラーが警告を出してくれます。

Generics に関するコンパイラーの問題

コンパイラーは、Generics の定義がされていないコレクションを使用するクラスをコンパイルするときにも、リスト 1 のコードでの場合と同じように警告を出します。例えば、以下の行が含まれるクラスをコンパイルするとします。

```
List buttonList = new LinkedList();
```

すると、コンパイラーは以下の警告を出します。

```
>javac GetIt.java
Note: GetIt.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

この警告は無視して構いません。誤って正しくないデータ型をコレクションに追加することがなければ、すべて問題なく運びます。

詳細な警告の表示

コンパイラーが警告している特定の問題を明らかにするには、`-Xlint:unchecked` コマンドをコンパイラーに渡してください。すると、リスト 3 のような出力が表示されます。

リスト 3. Xlint によって表示されたコンパイラーからの詳細情報

```
>javac -Xlint:unchecked GetIt.java
GetIt.java:7: warning: [unchecked] unchecked call to add(E) as a member of
        the raw type java.util.List
    buttonList.add(new JButton("One"));
                   ^
GetIt.java:8: warning: [unchecked] unchecked call to add(E) as a member of
        the raw type java.util.List
    buttonList.add(new JButton("Two"));
                   ^
GetIt.java:9: warning: [unchecked] unchecked call to add(E) as a member of
        the raw type java.util.List
    buttonList.add(new JButton("Three"));
                   ^
GetIt.java:10: warning: [unchecked] unchecked call to add(E) as a member of
        the raw type java.util.List
    buttonList.add(new JButton("Four"));
                   ^
4 warnings
```

リスト 3 を見るとわかるように、コンパイラーが問題としているのは `List` にデータ型が定義されていないことではありません。この出力が示しているのは、`List` にデータ型が定義されていないために、`add()` を呼び出すたびに問題が発生するということです。

この場合も警告なので無視して構いませんが、コレクションを修正して型を明示的に指定すれば、コンパイル時に発生した本当のエラーをこれらの警告に紛れて見落としてしまう可能性は低くなります。

コンパイラーからの警告の抑制

使用しているライブラリーを変更できない場合、あるいは変更したくない場合には、コンパイラーからの警告を抑制するという方法があります。`@SuppressWarnings` アノテーションはコンパイラーに対し、コードが警告を生成することはわかっているため、警告の表示は不要であると指示します。警告 [unchecked] が出ても無視する対象のメソッドの直前に以下の行を記述しておくと、コンパイラーはそのメソッドに関する警告を出さなくなります。

```
@SuppressWarnings("unchecked")
```

これで、クラスをコンパイルしたときに警告メッセージもエラー・メッセージも表示されなくなります。ただし、要求されている以外のデータ型が処理される場合に `ClassCastException` が発生するリスクがあることには変わりありません。この方法を使うかどうかは、その人次第です。

Web ページの読み取り

Generics を使うとどんな利点があるのか、そして Generics によってプログラムが管理しやすくなる仕組みについて十分に理解したところで、特定の Web ページ上のリンクをすべて収集するプログラムを作成してみます。読者の皆さんは Web ページを読み込んでコンテンツを解析するプログラムを自力で作成できるとは思いますが、その必要はありません。この機能は、Swing コンポーネント・ライブラリーに用意されているからです。したがって、必要となる作業はページ上のアンカー (`<a>`) タグに関連付けられたすべての `href` 属性を要求することのみとなります。

文書の取得

`javax.swing.text.html` パッケージに含まれている `HTMLEditorKit` にストリームを提供すると、関連付けられた Web ページが解析されます。解析されたストリームからこのキットに対して、すべての使用可能なタグを繰り返し処理してアンカー・タグの `href` 属性を取得するように指示することができます。このプログラムをより手の込んだものにして画像タグや Flash ムービーを収集することもできますが、このサンプルでは `<a href="...">xxx</a>` 形式のタグだけを収集することにします。

必要な作業は、新規 `HTMLEditorKit` インスタンスを作成して `Reader` をコンテンツに渡すことのみです。狙いはリモート Web サイトからコンテンツを取得することなので、この `Reader` にはコマンドラインに入力した `http://` ストリングを付けなければなりません。このストリングが `URL` コンストラクターに渡され、そこから `URLConnection` を受け取ることになります。複雑そうに聞こえるプロセスですが、そんなことはまったくありません。リスト 4 に、その方法を示します。

リスト 4. Web ページの読み取り

```
HttpURLConnection.setFollowRedirects(false);
EditorKit kit = new HTMLEditorKit();
Document doc = kit.createDefaultDocument();
doc.putProperty("IgnoreCharsetDirective", Boolean.TRUE);

String uri = args[0];
Reader reader = null;

if (uri != null && uri.startsWith("http")) {
  URLConnection conn = new URL(uri).openConnection();
  reader = new InputStreamReader(conn.getInputStream());
} else {
  System.err.println(
   "Usage: java ListUrls http://example.com/startingpage");
  System.exit(-1);
}

kit.read(reader, doc, 0);
```

接続に関連付けられた入力ストリームは、`EditorKit` の `read()` メソッドに渡されます。`read()` に渡されるその他の引数は、このキットの `createDefaultDocument()` を呼び出して作成する

Document、そして読み出しの開始位置です。この位置は通常、ストリームの始まりを示す 0 に設定されます。

リスト 4 では、有効な処理を 2 つ追加しています。その 1 つは、HttpURLConnection クラスの setFollowRedirects() メソッドを呼び出すというもので、以降のリダイレクト要求を無効にしています。そしてもう 1 つの処理では、Document の IgnoreCharsetDirective プロパティーをセットしています。これは、charset 属性がページの <meta> タグに含まれている場合の HTMLEditorKit の処理にバグがあることは明らかなため、その対策です。

要素の繰り返し処理

次に使用する Swing クラスは、同じく javax.swing.text パッケージに含まれている ElementIterator です。たった今作成したような Document の場合、そこに含まれる要素を繰り返し処理することができます。

```
ElementIterator it = new ElementIterator(doc);
javax.swing.text.Element elem;
while ((elem = it.next()) != null) {
  // ...
}
```

<a> タグを探すと関連付けられた href 属性がわかるので、これらの属性を検出されたリンクのコレクションに追加します。ここで使用しているコレクションが Set である理由は、重複して収集する必要がないからです。

```
Set<String> uriList = new TreeSet<String>();
// Below is inside of while loop
AttributeSet s = (AttributeSet)
  elem.getAttributes().getAttribute(HTML.Tag.A);
if (s != null) {
  String href = (String)
    s.getAttribute(HTML.Attribute.HREF);
  uriList.add(href);
}
```

すべてのリンクを収集するにはこれまで実行したステップで十分なものの、読者の皆さんは特殊なケースも扱えるようにしたいと思うかもしれません。しかし、例えば見つかった href がヌルであるというケースは追加する必要はありません (これは、整形式文書では起こるべきことではありませんが、たまにそうなることがあります)。一方、内部リンクには先頭に http:// が付いていないので、(次のタスクでも同じように) リストを再度ループしなければならない場合に備えて、これらの内部リンクを文書の基本 URL に続ける形にし、完全な URL を用意しておくのが最善です。さらに、javascript: タグには続けないほうが懸命かもしれません。この他にも多くの機能強化ができるはずです。リスト 5 に、完全なプログラムを記載します。

リスト 5. ある Web ページに含まれる URL をリストアップするためのコード

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class ListUrls {
  public static void main(String args[]) throws Exception {
```

```
    Set<String> uriList = new TreeSet<String>();
    HttpURLConnection.setFollowRedirects(false);
    EditorKit kit = new HTMLEditorKit();
    Document doc = kit.createDefaultDocument();
    doc.putProperty("IgnoreCharsetDirective", Boolean.TRUE);

    String uri = args[0];
    Reader reader = null;

    if (uri != null && uri.startsWith("http")) {
      URLConnection conn = new URL(uri).openConnection();
      reader = new InputStreamReader(conn.getInputStream());
    } else {
      System.err.println(
        "Usage: java ListUrls http://example.com/startingpage");
      System.exit(-1);
    }

    kit.read(reader, doc, 0);
    ElementIterator it = new ElementIterator(doc);
    javax.swing.text.Element elem;
    while ((elem = it.next()) != null) {
      AttributeSet s = (AttributeSet)
        elem.getAttributes().getAttribute(HTML.Tag.A);
      if (s != null) {
        String href = (String)s.getAttribute(HTML.Attribute.HREF);
        if (href == null) {
          continue;
        } else if (href.startsWith("javascript:")) {
          continue; // skip it
        } else if (href.startsWith("https:")) {
          // add as is
        } else if (!href.startsWith("http:")) {
          href = uri + href;
        }
        uriList.add(href);
      }
    }
    for (String element: uriList) {
      System.out.printf(">>%s<<%n", element);
    }
  }
}
```

このプログラムは収集した一連の URL を出力します。`ListUrls` プログラムをダウンロードしてコンパイルし、コマンドライン上でいずれかの URL に対してこのプログラムを実行してください (この記事の完全なソース・コードへのリンクは、「ダウンロード」セクションに記載されています)。実行結果は収集対象とするページによって異なります。

スレッド・プール

リスト 5 の `ListUrls` プログラムは特定のページから外部へのすべてのリンクを収集します。このプログラムを拡張してサイト全体をクロールさせるようにするには、プログラムを複数のタスクに分けるのが最善の方法です。ジョブを分割せずに単一のスレッドに維持することも可能ですが、そうすると、アプリケーションはすべての Web ページを読み取ってからでないとページを処理できなくなるため、I/O 遅延によってアプリケーションのブロック状態が発生してしまいます。さらに、複数のスレッドに分割する理由としてはネットワーク遅延もあります。`Set` の各要素をそれぞれのスレッドで処理すれば、ジョブ全体が大幅に高速化されるはずです。当然、スレッドの数は制限しなければなりません。さもないと、過剰な数のさまざまなタスクが実行され、実際の作業よりもタスクの切り替えに多くの時間が取られてしまうことになります。

Executor

Java SE 5 では、Generics の他、`java.util.concurrent` ライブラリーも導入されました (「参考文献」を参照)。`Executor` インターフェースは `Runnable` オブジェクトを受け取って実行します。この点は `Runnable` オブジェクトを `Thread` コンストラクターに渡すこととまったく同じですが、`Executor` では `Runnable` の取得後に再び `Thread` を使用して新しい `Runnable` を取得できるため、プログラムが絶えずスレッドを破棄しては作成しなおすというオーバーヘッドがなくなります。`Executor` インターフェースが持つメソッドは、`Runnable` パラメーターを受け取る `execute()` メソッドのみです。このメソッドの実行内容は、`Executor` インターフェースそれぞれの実装によって異なります。

`Executor` の実装の一例には、`ThreadPoolExecutor` があります。スレッド・プールを作成するために `ThreadPoolExecutor` コンストラクターを直接呼び出すことはありませんが、代わりに `Executors` ユーティリティー・クラスを使用してスレッド・プールを作成します。固定サイズのスレッド・プールの場合には `newFixedThreadPool(int maxThreads)` を使用してください。あるいは `newFixedThreadPool(int maxThreads, ThreadFactor factory)` を使用して、基礎となるスレッドを作成するためのファクトリーを提供することもできます。

スレッド・プールを作成したら、`service(Runnable)` メソッドを使って、実行するタスクを追加します。ここで作成する Web クローラーの場合、`awaitTermination()` メソッドを呼び出すと、すべてのタスクがいつ完了するか、あるいは少なくともスレッド・プール内のタスクがいつ完了するかはわかります (リスト 6 を参照)。

リスト 6. スレッド・プールの操作

```
String uri =...
ExecutorService service = Executors.newFixedThreadPool(5);
service.execute(new Crawler(service, uri, uri));
service.awaitTermination(300, TimeUnit.SECONDS);
for (String element: allUriList) {
  System.out.printf(">>%s<<%n", element);
}
```

`awaitTermination()` メソッドはタイムアウトを許可します。リスト 6 の場合、プログラムがタイムアウトするように設定されているのは 5 分後ですが、プログラムを実行させる時間、ネットワーク接続の速度、そしてクロールする Web サイトの深さに応じて、タイムアウトをこれより短くすることも、長くすることもできます。

クローラーには基本 URI ストリングしか追加されていない点にも注目してください。各ページが読み込まれるごとに、新しい URI がジョブ・キューに追加されていきます。

Runnable

リスト 5 に記載したコードの大半は、サービスが実行する `Runnable` タスクです。私はいくつかチェックを追加して、次のページで収集する URL のリストの作成方法を改善しました。`execute()` メソッドの終了時のチェックは、サービスを終了すべきかどうかを調べるためのものです。プールはプログラムが終了するまで実行されるのが通常ですが、このプログラムはプール内のタスクが完了した時点で終了するため、このチェックが必要となります。

`CollectUrls` プログラムをダウンロードし、比較的小さなサイト (できれば自分のサイト) で実行して、そのサイトからすべてのリンクのダンプを取得してみてください。このプログラムは、マルチマップを維持するように変更することも可能です。つまり、各リンクのソースがわかっている場合、サイトの階層と相互接続のマップを自動的に生成することができます。

その他のスレッド・プール

`CollectUrls` Web クローラー・プログラムは固定サイズのスレッド・プールを利用しますが、使用できるスレッド・プールはこれだけではありません。`Executors` ユーティリティー・クラスを使用して作成できるスレッド・プールには、他にも 3 つのタイプがあります。

- **`newCachedThreadPool()`** は、潜在的にサイズの制限がないプールを作成しますが、スレッドが長い間アイドル状態になるとそのスレッドをキルします。このクラスは、短時間の非同期タスクが多数ある場合に使用してください。プール内に使用可能なスレッドがあれば、そのスレッドが使用され、使用可能なスレッドがなければ新しいスレッドが作成されます。そしてプール内で 60 秒間アイドル状態になったスレッドは消去されます。タスクが実行されていないときには、リソースは使用されません。固定サイズのプールはこれとは対照的に、実行するタスクがないときでも、すべてのスレッドを待機状態のままにします。
- **`newSingleThreadExecutor()`** は、順次完了する必要があるジョブに役立つプールを作成します。基礎となるスレッドが破棄されると、そのスレッドが再び作成されます。これは固定サイズのスレッド・プールを作成する場合と同様ですが、基礎となるプールのサイズを変更することはできません。
- **`newScheduledThreadPool()`** は、`Timer` オブジェクトのように機能するプールを作成しますが、このプールはキャッチされない例外とスレッド停止状態の処理に優れています。`Timer` クラスでは、長時間実行される `TimerTask` によって他のタスクの実行がブロックされる場合があります。プールに複数のスレッドを持つことで、そのような事態が起こらないようにするのと同時に、スレッドのスケジュールを維持することが可能になります。

使用できるコレクションには他のタイプもあります。例えば、スケジューリングされたスレッド・プールの代わりに `DelayQueue` を使用すると、遅延時間が過ぎるまで取り出すことができない項目をコレクションに追加することができます。このキューは `BlockingQueue` の特定のタイプで、項目がすぐに使用できない場合、遅延時間が過ぎるまでは、その項目はキューから取得できません。

まとめ

この記事では、以下の手順によって Web クローラーを作成するというタスクを説明しました。

- Generics の `Set` で検出された一連の URI ストリングを収集する
- `Runnable` タスクを実行し、サイトを構成するページでさらに URI を見つける
- スレッド・プールを使用して `Runnable` タスクを操作する

Web クローラーを拡張する方法としては、参照画像の収集や、特定のテキスト・ストリングの検索などが考えられます。このプログラムの機能をさまざまな方法で拡張して、並行コレクション手法の使い方をさらに詳しく学んでください。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-collections-code.zip	3KB

著者について

John Zukowski

Paul Duvall は、Stelligent Incorporated の最高技術責任者です。同社はアジャイル・コンサルタント会社として、開発チームが production-ready software を提供できるように支援しています。彼は Addison-Wesley Signature Series から出版されている『Continuous Integration: Improving Software Quality and Reducing Risk』(Addison-Wesley Professional、2007年、2008年度 Jolt Award を受賞) の共著者で、『UML 2 Toolkit』(Wiley、2003年) および『No Fluff Just Stuff Anthology』(Pragmatic Programmers、2007年) の著作にも貢献しました。