ヒント: 例外をスローできない場合

例外を使用しないスーパークラスを扱う

Elliotte Rusty Harold
Software Engineer
Cafe au Lait

2010年 4月 06日

今回のヒントでは、オーバーライドが必要なメソッドに、十分拡張できる `throws` 節がない場合に、インターフェースを実装する問題、つまりクラスをサブクラス化する問題について Elliotte Rusty Harold が検討します。そして、チェック例外を処理することもスローすることもできない場合の選択肢を探ります。

チェック例外の 1 つの問題は、チェック例外をスローすることが許されない場合があることです。特に、スーパークラスの中で宣言されたメソッドをオーバーライドする場合や、インターフェースの中で宣言されたメソッドを実装する場合、そのメソッドがチェック例外を宣言していなければ、実装でチェック例外を宣言することもできません。このため、前もって例外を処理しなければなりませんが、それには例外をランタイム例外に変換する方法や、あるいは例外の発生を抑えて例外を処理しないようにする方法などがあります。しかしこうした方法は果たして適切なのでしょうか。より深いところに何か問題はないのでしょうか。

問題

例を見てみると、この問題が明確になります。例えば `File` オブジェクトの `List` があり、それらのオブジェクトをオブジェクトの正規パス (つまりエイリアスやシンボリック・リンク、`/../` や `/./` を解決した後の完全な絶対パス) を基準に辞書の順序でソートしたいとします。単純な手法として、コンパレーターを使う方法があります (リスト 1)。

リスト 1. 正規パスを基準に 2 つのファイルを比較する

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class FileComparator implements Comparator<File> {

    public int compare(File f1, File f2) {
        return f1.getCanonicalPath().compareTo(f2.getCanonicalPath());
    }
```

```
    public static void main(String[] args) {
      ArrayList<File> files = new ArrayList<File>();
      for (String arg : args) {
          files.add(new File(arg));
      }
      Collections.sort(files, new FileComparator());
      for (File f : files) {
          System.out.println(f);
      }
    }

}
```

残念ながら、このコードをコンパイルすることはできません。`getCanonicalPath()` メソッドはファイルシステムにアクセスする必要があるため、`IOException` をスローしてしまうという問題があるからです。通常、チェック例外を処理する場合には、この問題に対して下記の 2 つの方法のいずれかで対応することができます。

1. 例外を発生させるコードを `try` ブロックでラップし、スローされる例外をすべてキャッチする。
2. 問題部分を包含するメソッド (この場合は `compare()`) も `IOException` をスローするものとして宣言する。

通常、どちらを選択するかは、例外がスローされた時点でその例外を適切に処理できるかどうかによります。適切に処理できるようであれば、`try-catch` ブロックを使います。適切に処理できないようであれば、問題部分を包含するメソッド自体が例外をスローするものとして宣言します。あいにく、この例ではどちらの方法も使えません。

`compare()` メソッドの中で `IOException` を適切に処理することはできません。技術的には、単純に `0` または `1` または `-1` を返すことで、適切に処理することもできなくはありません (リスト 2)。

リスト 2. 例外の発生時にデフォルト値を返す

```
public int compare(File f1, File f2) {
    try {
        return f1.getCanonicalPath().compareTo(f2.getCanonicalPath());
    }
    catch (IOException ex) {
       return -1;
    }
}
```

しかしこの方法は安定した結果が得られないため、`compare()` メソッドの契約に違反します。同じオブジェクトを使ってこのメソッドを 2 回呼び出すと、結果が異なる可能性があります。コンパレーターをソートに使うということは、結局リストは適切にソートされない、ということになります。そこで今度は第 2 の選択肢、`compare() throws IOException` を宣言する方法を試してみましょう。

```
public int compare(File f1, File f2) throws IOException {
    return f1.getCanonicalPath().compareTo(f2.getCanonicalPath());
}
```

これではコンパイルすることすらできません。チェック例外はメソッドのシグニチャーの一部であるため、オーバーライドしたメソッドの戻り型を変更することはできても、オーバーライドし

たメソッドにチェック例外を追加することはできません。そこで 1 番目と 2 番目のあいだの中間的な選択肢として `compare()` の中で例外をキャッチし、それをランタイム例外に変換します。そうすると、このランタイム例外をスローすることができます (リスト 3)。

リスト 3. チェック例外をランタイム例外に変換する

```
public int compare(File f1, File f2) {
    try {
        return f1.getCanonicalPath().compareTo(f2.getCanonicalPath());
    }
    catch (IOException ex) {
       throw new RuntimeException(ex);
    }
}
```

このコードをコンパイルすることはできますが、あいにくこの方法も適切ではありません。理由は先ほど以上に微妙です。`Comparator` インターフェースは契約を定義します (「参考文献」を参照)。この契約では、このメソッドがランタイム例外をスローすることを許可していません (呼び出し側のコード内のバグとなりうる一般的なタイプ・セーフ違反は禁止されています)。このコンパレーターを使うメソッドは、契約に従い、このコンパレーターを使って 2 つのファイルを比較しますが、例外をスローすることはありません。これらのメソッドは、`compare()` から発生する予期せぬ例外を処理するように作られることはないからです。

コードによって処理すべき外部条件に対してランタイム例外を使うのが不適切な理由は、まさにこの微妙な点にあります。ランタイム例外を使うと問題が隠されてしまい、その問題を実際に処理することができません。データの破損や不正な結果など、例外を処理しないことによる悪影響はすべて残ってしまいます。

そのためジレンマに陥ります。`compare()` の中で例外を処理することは現実的にはできず、`compare()` の外で例外を処理することもできません。ではどういった方法が残されているのでしょう。`System.exit()` でしょうか。唯一の適切なソリューションは、このジレンマを完全に回避することであり、幸いその回避方法には少なくとも 2 通りの方法があります。

問題を分割する

第 1 の方法では、問題を 2 つの部分に分解します。比較によって例外が発生することはありません。比較の対象は単なるストリングだからです。例外は、正規パスを用いてファイルをストリングに変換することによって発生します。例外をスローする可能性のある操作と、例外をスローする可能性のない操作とを分離すると、問題が扱いやすくなります。つまり最初にすべてのファイル・オブジェクトをストリングに変換し、次にそれらのストリングをストリング・コンパレーターによって (さらには `java.lang.String` による自然順序によって) ソートし、そして最後に、ソートされたストリング・リストを使って元のファイル・リストをソートするのです。この方法は少し間接的ですが、リストが変更される前に `IOException` をスローするという利点があります。例外が発生する場合には、何らかのダメージが与えられる前に、適切に定義されたポイントで例外が発生します。そして呼び出し側のコードは、その例外に対処する方法を判断することができます。リスト 4 はこの方法を示しています。

リスト 4. 最初に読み取り、次にソートする

```
import java.io.File;
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;

public class FileComparator {

    private static ArrayList<String> getCanonicalPaths(ArrayList<File> files)
            throws IOException {
        ArrayList<String> paths = new ArrayList<String>();
        for (File file : files) paths.add(file.getCanonicalPath());
        return paths;
    }

    public static void main(String[] args) throws IOException {
      ArrayList<File> files = new ArrayList<File>();
      for (String arg : args) {
          files.add(new File(arg));
      }

      ArrayList<String> paths = getCanonicalPaths(files);

      // to maintain the original mapping
      HashMap<String, File> map = new HashMap<String, File>();
      int i = 0;
      for (String path : paths) {
          map.put(path, files.get(i));
          i++;
      }

      Collections.sort(paths);
      files.clear();
      for (String path : paths) {
          files.add(map.get(path));
      }
    }

}
```

リスト 4 によって I/O エラーの可能性がなくなるわけではありません。I/O エラーが起こるかどうかは、このコードの外部で何が行われるかに左右されるため、I/O エラーの可能性をなくすことはできません。しかしこの方法によって、対応が容易な場所に問題を移すことができたのです。

問題を回避する

上で説明した方法は少し複雑なので、第 2 の方法を提案しましょう。この方法では、組み込みの `compare()` 関数または `Collections.sort()` をまったく使いません。compare() 関数や Collections.sort() を使うと便利ですが、こうした場合に使うのは適切ではないかもしれないということを考慮する必要があります。`Comparable` と `Comparator` は、比較演算を行うことが決まっている場合や予測される場合のために作られています。その前提は、I/O が入り込むことで崩れてしまいます。そうした場合には、通常のアルゴリズムやインターフェースを適用できない可能性が高くなります。通常のアルゴリズムやインターフェースが使えるとしても、非常に効率の悪いものになる可能性があります。

例えば、正規パスによってファイルを比較する代わりにファイルの内容によって比較を行うとします。比較演算ごとに比較対象の 2 つのファイルの内容を読み取る必要があり、場合によっては 2 つのファイルの全内容を読み取る必要があるかもしれません。そうした場合、効率的なアルゴリズムであれば読み取りの回数を最小限にしようとし、ファイルを比較するごとに各ファイルを

再度読み込むのではなく、各読み込みの結果をキャッシュに入れようとするかもしれません。あるいはファイルが大きな場合には、ファイルのハッシュコードをキャッシュしようとするかもしれません。この場合も、考え方としては最初に比較用のキー・リストにキーを追加してからソートを行うようにしており、インラインでソートしているわけではありません。

別途、並列に `IOComparator` インターフェースを定義し、このインターフェースを使って必要な例外をスローする方法が考えられます (リスト 5)。

リスト 5. 独立している `IOComparator` インターフェース

```
import java.io.IOException;

public interface IOComparator<T> {

    int compare(T o1, T o2) throws IOException;

}
```

次に、このクラスをベースとして別途並列に一連のユーティリティーを定義し、それらのユーティリティーによってコレクションの一時コピーに対して必要な処理を行います。そうすれば、それらのユーティリティーは、破損する可能性のある中間的な状態にデータ構造を放置することなく例外をスローすることができます。例えば、リスト 6 は基本的なバブル・ソートを示しています。

リスト 6. ファイルをバブル・ソートする

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class IOSorter {

    public static <T> void sort(List<T> list, IOComparator<? super T> comparator)
      throws IOException {
        List<T> temp = new ArrayList<T>(list.size());
        temp.addAll(list);

        bubblesort(temp, comparator);

        // copy back to original list now that no exceptions have been thrown
        list.clear();
        list.addAll(temp);
    }

    // of course you can replace this with a better algorithm such as quicksort
    private static <T> void bubblesort(List<T> list, IOComparator<? super T> comparator)
      throws IOException {
        for (int i = 1; i < list.size(); i++) {
            for (int j = 0; j < list.size() - i; j++) {
                if (comparator.compare(list.get(j), list.get(j + 1)) > 0) {
                    swap(list, j);
                }
            }
        }
    }

    private static <T> void swap(List<T> list, int j) {
        T temp = list.get(j);
        list.set(j, list.get(j+1));
```

```
            list.set(j + 1, temp);
        }

    }
```

これが唯一の方法というわけではありません。念のため説明すると、リスト 6 は既存の `Collections.sort()` メソッドと意図的に似たものにしていますが、古いリストを変更するよりも新しいリストを返した方が適切かもしれません。まさにそうすることで、変更の最中に例外がスローされることで発生しうる問題を回避することができます。

これで I/O エラーが発生する可能性を実際に認識し、そのエラーに対応できるようになったので、I/O エラーを隠す代わりに、さらに高度なエラー訂正を行うことができます。例えば、`IOComparator` が I/O エラーを取り込めなかったとしても、I/O に関する多くの問題は変わりやすいので、何回か繰り返して試行するという方法を採ることができます (リスト 7)。

リスト 7. 最初に成功しなかった場合には、何度も (ただし膨大な回数ではなく) 繰り返し試す

```
import java.io.File;
import java.io.IOException;

public class CanonicalPathComparator implements IOComparator<File> {

    @Override
    public int compare(File f1, File f2) throws IOException {
        for (int i = 0; i < 3; i++) {
            try {
                return f1.getCanonicalPath().compareTo(f2.getCanonicalPath());
            }
            catch (IOException ex) {
                continue;
            }
        }
        // last chance
        return f1.getCanonicalPath().compareTo(f2.getCanonicalPath());
    }

}
```

この方法によって通常の `Comparator` の問題を解決することはできません。例外をスローしないためには無限に再試行する必要がありますが、I/O には一時的ではない問題が数多くあるからです。

チェック例外の考え方は不適切なのか

`java.io.IOException` がチェック例外ではなくランタイム例外だったとしたら、この記事で説明した内容は何か変わるのでしょうか。その答えは「イエス」です。`IOException` が `java.lang.Exception` を継承する代わりに `RuntimeException` を継承したとすると、実行時に I/O エラーが実際に発生する可能性を無視して予期せぬ失敗をするような、バグの多い不適切なコードを作成しやすくなります。

しかし、I/O エラーに対する準備を整え、I/O エラーを処理する適切なコードを作成するのは容易ではありません。想定外の I/O エラーは決して起こらないものとし、I/O エラーへの対応も考慮する必要がない方法に比べ、この記事で説明した方法は確かに複雑です。しかし Java 言語からチェック例外をなくしたとしても、問題が解決されるわけではありません。I/O エラーや他の環境

上の問題は現実であり、こうした問題を無視するよりも、こうした問題に備えた方がはるかに得策です。

結局のところ、チェック例外がメソッド・シグニチャーの一部であることには十分な理由があるのです。チェック例外をスローしてはならないメソッドからチェック例外をスローしようとしている (従って発生するのを抑えてはならない例外を抑止している) ことに気付いた場合には、そこで立ち止まって見直しを行い、そもそもなぜそのメソッドをオーバーライドしようとしているのかを考えてみることです。そうすれば、まったく別のことをすべきことに気付くことが多いはずです。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold が初めて Java 言語を学んだのは 1995年のことです。それ以前は、最初に Fortran、次に Applesoft Basic を使用していました。C はおそらく彼の 3 番目の言語で、4 番目は Microphone II だったかもしれません。5 番目は Pascal でしたが、それほど深くこの言語に関わることはありませんでした。6 番目はおそらく IDL (Interactive Data Language)、7 番目は Perl でしょうか。Java は彼が学んだ 8 番目の言語で、他のどの言語より没頭した言語です。けれどもその後も、彼は新しい言語として PHP、AppleScript (Java より古いかもしれません)、XSLT、XQuery、C++、そして最近では Haskell を学んでいます。