関数型の考え方: Either と Option による関数型のエラー処理

タイプ・セーフな関数型の例外

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 7月 12日

Java 開発者が慣れているエラー処理の方法は、例外をスローしてキャッチするというものですが、その方法は、関数型のパラダイムにはマッチしません。連載「関数型の考え方」の今回の記事では、コードをタイプ・セーフな状態に維持しつつ、関数型の方法で Java エラーを示す方法を調査します。さらに、関数型の return文でチェック例外をラップする方法を説明し、`Either` という便利な抽象クラスを紹介します。

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。

関数型プログラミングのような奥の深い話題について調査していると、それに関連して興味をそそられる別の話題が持ち上がってくることがよくあります。前回の記事では連載のミニシリーズの続きとして、従来の Gang of Four (GoF) のデザイン・パターンを関数型の手法で再考しました。デザイン・パターンについては、次回の記事で検討する Scala スタイルのパターン・マッチングで再び取り上げますが、その前に、`Either` と呼ばれる概念を通してある程度の基礎知識を学んでおく必要があります。`Either` の用途の 1 つは、この記事で説明する、関数型スタイルのエラー処理です。今回の記事で皆さんに `Either` による魔法のようなエラー処理を理解してもらった上で、次回の記事でパターン・マッチングとツリーを取り上げる予定です。

Java でエラーを処理するために従来使用してきた手段は、例外と、例外を作成して伝播するための Java 言語のサポートです。けれども、構造化例外処理がなかったとしたらどうしますか？関数型言語の多くでは、構造化という例外パラダイムをサポートしていないため、エラー条件

を表現する別の方法を見つけなければなりません。この記事では、Java で通常の例外伝播メカニズムを使用せずにエラーを処理する、タイプ・セーフなメカニズムを紹介します (一部の例では、Functional Java フレームワークの助けを借ります)。

関数型のエラー処理

Java で例外を使わずにエラーを処理しようとするときに根本的な障害となるのは、Java 言語ではメソッドからの戻り値が 1 つに限られることです。ただし、複数の値を保持することが可能な単一の `Object` (またはサブクラス) 参照を返すことは当然可能です。そうすれば、`Map` を使用して複数の戻り値を処理することができます。一例として、リスト 1 に記載する `divide()` メソッドを見てください。

リスト 1. `Map` を使用して複数の戻り値を処理する

```
public static Map<String, Object> divide(int x, int y) {
    Map<String, Object> result = new HashMap<String, Object>();
    if (y == 0)
        result.put("exception", new Exception("div by zero"));
    else
        result.put("answer", (double) x / y);
    return result;
}
```

リスト 1 で私が作成しているのは、キーを `String` に設定し、値を `Object` に設定した `Map` です。`divide()` メソッドには、失敗を示す `exception`、または成功を示す `answer` のいずれかが入ります。リスト 2 で、両方の場合をテストします。

リスト 2. `Map` を使用して成功および失敗の処理をテストする

```
@Test
public void maps_success() {
    Map<String, Object> result = RomanNumeralParser.divide(4, 2);
    assertEquals(2.0, (Double) result.get("answer"), 0.1);
}

@Test
public void maps_failure() {
    Map<String, Object> result = RomanNumeralParser.divide(4, 0);
    assertEquals("div by zero", ((Exception) result.get("exception")).getMessage());
}
```

リスト 2 の `maps_success` は、返された `Map` の中に正しいエントリーが存在することを検証するテストです。`maps_failure` は、例外ケースをチェックするテストです。

この手法には、明らかな問題がいくつかあります。第 1 に、最終的な `Map` はいずれにしてもタイプ・セーフではないため、コンパイラーが特定のエラーを捕捉できなくなります。列挙をキーとして使用すれば、この問題は多少改善されるとは言え、それほど役には立ちません。第 2 に、メソッドの呼び出し側はメソッド呼び出しが成功したかどうかがわからないため、想定され得る結果をチェックするという負担が呼び出し側にかかります。第 3 に、両方のキーに値が設定されて結果が曖昧になってしまうことを防ぐための手段がまったく講じられていません。

そこで必要になるのが、タイプ・セーフな方法で 2 つ (またはそれ以上) の値を返すためのメカニズムです。

`Either` クラス

関数型言語では、2 つの別個の値を返さなければならないという事態が頻繁に起こります。この振る舞いをモデル化するために使用される共通のデータ構造が、`Either` クラスです。Java では Generics を使用して、リスト 3 のような単純な `Either` クラスを作成することができます。

リスト 3. `Either` クラスを介して 2 つの (タイプ・セーフな) 値を返す

```
public class Either<A,B> {
    private A left = null;
    private B right = null;

    private Either(A a,B b) {
        left = a;
        right = b;
    }

    public static <A,B> Either<A,B> left(A a) {
        return new Either<A,B>(a,null);
    }

    public A left() {
        return left;
    }

    public boolean isLeft() {
        return left != null;
    }

    public boolean isRight() {
        return right != null;
    }

    public B right() {
        return right;
    }

    public static <A,B> Either<A,B> right(B b) {
        return new Either<A,B>(null,b);
    }

   public void fold(F<A> leftOption, F<B> rightOption) {
        if(right == null)
            leftOption.f(left);
        else
            rightOption.f(right);
    }
}
```

リスト 3 では、`Either` には `left` または `right` のいずれか一方の値が保持されるように作られています (両方の値が保持されることは決してありません)。このようなデータ構造は、非交和 (disjoint union) と呼ばれます。C ベースの一部の言語には、複数の異なる型のうち 1 つのインスタンスを格納できる共用体 (`union`) というデータ型があります。非交和には 2 つの型のスロットがありますが、保持されるのはどちらか一方の型のインスタンスだけです。`Either` クラスは `private` コンストラクターを使用して、インスタンスの作成を 2 つの静的メソッド、`left(A a)` または `right(B b)` のいずれかに任せます。Either クラスに含まれる残りのメソッドは、クラス・メンバーを取得したり、調査したりするためのヘルパーです。

`Either` という手段があれば、タイプ・セーフを確保すると同時に、例外または正当な結果のどちらか一方 (決して両方ということはありません) だけを返すコードを作成することができます。一

般的な関数型の慣例として、`Either` クラスの left には例外 (例外が発生した場合) を格納し、right には結果を格納します。

ローマ数字を解析する

Either を使用する例を示すために、私が作成した `RomanNumeral` という名前のクラス (その実装については、読者のご想像にお任せします) と、この `RomanNumeral` クラスを呼び出す `RomanNumeralParser` という名前のクラスを用います。リスト 4 に、`parseNumber()` メソッドとこのメソッドを説明するためのテストを記載します。

リスト 4. ローマ数字を解析する

```
public static Either<Exception, Integer> parseNumber(String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return Either.left(new Exception("Invalid Roman numeral"));
    else
        return Either.right(new RomanNumeral(s).toInt());
}

@Test
public void parsing_success() {
    Either<Exception, Integer> result = RomanNumeralParser.parseNumber("XLII");
    assertEquals(Integer.valueOf(42), result.right());
}

@Test
public void parsing_failure() {
    Either<Exception, Integer> result = RomanNumeralParser.parseNumber("FOO");
    assertEquals(INVALID_ROMAN_NUMERAL, result.left().getMessage());
}
```

リスト 4 の `parseNumber()` メソッドは、(エラーを明らかにするための) 驚くほど単純な検証を行い、エラー条件が見つかるとそれを `Either` の left に格納し、結果が正常に得られれば、その結果を right に格納します。このユニット・テストには、両方の場合が示されています。

この方法は、`Map` を渡す方法と比べると大幅に改善されています。まず、タイプ・セーフが確保されます (例外はいくらでも具体的なものにできることに注目してください)。エラーは、Generics によるメソッド宣言で明らかです。また、結果はいったん間接的な手段で返されて、`Either`から (例外または答えとして) アンパックされます。そして、この間接的な手段が、遅延を可能にします。

遅延解析と Functional Java

多くの関数型アルゴリズムに登場する `Either` クラスは、関数型の世界ではあまりにもよく使用されているため、Functional Java フレームワーク (「参考文献」を参照) にはリスト 3 とリスト 4 の例でも機能する Either の実装が含まれています。ただし、この `Either` 実装は Functional Java の他の構成体と連動するように作成されています。したがって、遅延エラー評価を作成するには、`Either` と Functional Java の `P1` クラスを組み合わせて使用します。遅延式は、必要になったときに実行されます (「参考文献」を参照)。

Functional Java の `P1` クラスは、パラメーターを取らない `_1()` と名付けられたメソッドのみをラップする単純なラッパーです (`P2`、`P3` などの他のバリエーションは、複数のメソッドを包含します)。Functional Java での `P1` の用途は、コード・ブロックを実行することなく渡し、任意のコンテキストでそのコードを実行できるようにすることです。

Java では、例外をスローすると同時にその例外がインスタンス化されますが、遅延評価されるメソッドを返すことで、例外の作成を延期することができます。リスト 5 のサンプル・コードおよび関連するテストを見てください。

リスト 5. Functional Java を使用して遅延パーサーを作成する

```
public static P1<Either<Exception, Integer>> parseNumberLazy(final String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return new P1<Either<Exception, Integer>>() {
            public Either<Exception, Integer> _1() {
                return Either.left(new Exception("Invalid Roman numeral"));
            }
        };
    else
        return new P1<Either<Exception, Integer>>() {
            public Either<Exception, Integer> _1() {
                return Either.right(new RomanNumeral(s).toInt());
            }
        };
}

@Test
public void parse_lazy() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.parseNumberLazy("XLII");
    assertEquals((long) 42, (long) result._1().right().value());
}

@Test
public void parse_lazy_exception() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.parseNumberLazy("FOO");
    assertTrue(result._1().isLeft());
    assertEquals(INVALID_ROMAN_NUMERAL, result._1().left().value().getMessage());
}
```

リスト 5 のコードは、`P1` ラッパーを追加したという点を除けば、リスト 4 のコードと同様です。`parse_lazy` テストでは、結果で `_1()` メソッドを呼び出すことによって、結果をアンパックする必要があります。それにより、`Either` の right が返され、そこから値を取得することができます。`parse_lazy_exception` テストでは、left の有無を確認し、これが存在する場合には例外をアンパックして、そのメッセージを判別することができます。

`_1()` メソッドの呼び出しで `Either` の left をアンパックするまでは、例外 (そして、生成するのにコストがかかるスタック・トレース) は作成されません。したがって、例外は遅延され、例外のコンストラクターの実行を遅らせることができます。

デフォルト値を提供する

`Either` をエラー処理に使用するメリットは、遅延だけではありません。デフォルト値を提供できるというメリットもあります。リスト 6 のコードを見てください。

リスト 6. 妥当なデフォルトの戻り値を提供する

```
public static Either<Exception, Integer> parseNumberDefaults(final String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return Either.left(new Exception("Invalid Roman numeral"));
    else {
        int number = new RomanNumeral(s).toInt();
        return Either.right(new RomanNumeral(number >= MAX ? MAX : number).toInt());
    }
}

@Test
public void parse_defaults_normal() {
    Either<Exception, Integer> result = FjRomanNumeralParser.parseNumberDefaults("XLII");
    assertEquals((long) 42, (long) result.right().value());
}

@Test
public void parse_defaults_triggered() {
    Either<Exception, Integer> result = FjRomanNumeralParser.parseNumberDefaults("MM");
    assertEquals((long) 1000, (long) result.right().value());
}
```

リスト 6 では、`MAX` より大きいローマ数字は決して許可しないという前提となっており、`MAX` より大きいローマ数字を設定しようとすると、デフォルトで MAX の値が設定されます。`parseNumberDefaults()` メソッドによって、`Either` の right には MAX より大きい値の代わりに必ずデフォルト値が置かれるようにしています。

例外をラップする

`Either` を使用して例外をラップすることもできます。こうすることにより、構造化例外処理を関数型の例外処理に変換することができます。リスト 7 を参照してください。

リスト 7. 他の例外をキャッチする

```
public static Either<Exception, Integer> divide(int x, int y) {
    try {
        return Either.right(x / y);
    } catch (Exception e) {
        return Either.left(e);
    }
}

@Test
public void catching_other_people_exceptions() {
    Either<Exception, Integer> result = FjRomanNumeralParser.divide(4, 2);
    assertEquals((long) 2, (long) result.right().value());
    Either<Exception, Integer> failure = FjRomanNumeralParser.divide(4, 0);
    assertEquals("/ by zero", failure.left().value().getMessage());
}
```

リスト 7 で私が試みているのは、`ArithmeticException` を発生する可能性のある除算です。例外が発生した場合には、その例外を `Either` の left 内にラップします。例外が発生しなければ、right で結果を返します。このように、`Either` を使用すれば、従来の例外 (チェック例外を含む) を関数型に近いスタイルに変換することができます。

もちろん、呼び出したメソッドからスローされた例外を遅延してラップすることもできます (リスト 8 を参照)。

Listing 8. Lazily catching exceptions

```
public static P1<Either<Exception, Integer>> divideLazily(final int x, final int y) {
    return new P1<Either<Exception, Integer>>() {
        public Either<Exception, Integer> _1() {
            try {
                return Either.right(x / y);
            } catch (Exception e) {
                return Either.left(e);
            }
        }
    };
}

@Test
public void lazily_catching_other_people_exceptions() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.divideLazily(4, 2);
    assertEquals((long) 2, (long) result._1().right().value());
    P1<Either<Exception, Integer>> failure = FjRomanNumeralParser.divideLazily(4, 0);
    assertEquals("/ by zero", failure._1().left().value().getMessage());
}
```

例外をネストする

Java の例外処理に備わっている便利な特徴の 1 つは、複数の異なる潜在的例外タイプをメソッド・シグニチャーの一部として宣言できることです。`Either` でもそれは可能ですが、その場合、構文がますます複雑になってきます。例えば、`RomanNumeralParser` のメソッドとして、2 つのローマ数字による除算を行い、2 つの異なる例外条件 (構文解析エラーまたは除算エラー) を返さなければならないとしたら、どうすればよいのでしょう？標準的な Java の Generics を使用すれば、リスト 9 のような方法で例外をネストすることができます。

リスト 9. ネストされた例外

```
public static Either<NumberFormatException, Either<ArithmeticException, Double>>
        divideRoman(final String x, final String y) {
    Either<Exception, Integer> possibleX = parseNumber(x);
    Either<Exception, Integer> possibleY = parseNumber(y);
    if (possibleX.isLeft() || possibleY.isLeft())
        return Either.left(new NumberFormatException("invalid parameter"));
    int intY = possibleY.right().value().intValue();
    Either<ArithmeticException, Double> errorForY =
            Either.left(new ArithmeticException("div by 1"));
    if (intY == 1)
        return Either.right((fj.data.Either<ArithmeticException, Double>) errorForY);
    int intX = possibleX.right().value().intValue();
    Either<ArithmeticException, Double> result =
            Either.right(new Double((double) intX) / intY);
    return Either.right(result);
}

@Test
public void test_divide_romans_success() {
    fj.data.Either<NumberFormatException, Either<ArithmeticException, Double>> result =
        FjRomanNumeralParser.divideRoman("IV", "II");
    assertEquals(2.0,result.right().value().right().value().doubleValue(), 0.1);
}

@Test

public void test_divide_romans_number_format_error() {
    Either<NumberFormatException, Either<ArithmeticException, Double>> result =
        FjRomanNumeralParser.divideRoman("IVooo", "II");
    assertEquals("invalid parameter", result.left().value().getMessage());
}
```

```
@Test
public void test_divide_romans_arthmetic_exception() {
    Either<NumberFormatException, Either<ArithmeticException, Double>> result =
        FjRomanNumeralParser.divideRoman("IV", "I");
    assertEquals("div by 1", result.right().value().left().value().getMessage());
}
```

リスト 9 では、`divideRoman()` メソッドがまず、リスト 4 で使用した元の `parseNumber()` メソッドから返された `Either` をアンパックします。2 つの数値変換のいずれかで例外が発生した場合、`Either` の left で例外を返します。次に、実際の整数値をアンパックしてから、別の検証基準を実行する必要があります。ローマ数字にはゼロ (0) の概念がないため、1 による除算を許可しないルールを作りました。分母が 1 の場合には、例外をパッケージ化して、right にネストされた left にそのパッケージを配置します。

別の言葉に置き換えると、`NumberFormatException`、`ArithmeticException`、および `Double` という 3 つの型で表現された 3 つのスロットがあるということです。最初の `Either` は、left に潜在的 `NumberFormatException` を格納し、right に別の `Either` を格納します。2 番目の `Either` は、left に潜在的 `ArithmeticException` を格納し、right にペイロード、つまり結果を格納します。したがって、実際の答えを得るには、`result.right().value().right().value().doubleValue()` をトラバースしなければならないということです！この手法に実用性がないことはすぐにわかりますが、クラス・シグニチャーの一部として例外をネストするにはタイプ・セーフな方法となります。

`Option` クラス

`Either` という便利な概念は、次回の記事でツリー状のデータ構造を作成する際に使用します。Scala には、この Either と似たような `Option` というクラスがあります (Option は Functional Java にもあります)。このクラスは、より単純な例外ケースとして、none で正当な値がないことを示し、some で正常な戻り値があることを示します。リスト 10 に、`Option` の例を記載します。

リスト 10. `Option` を使用する

```
public static Option<Double> divide(double x, double y) {
    if (y == 0)
        return Option.none();
    return Option.some(x / y);
}

@Test
public void option_test_success() {
    Option result = FjRomanNumeralParser.divide(4.0, 2);
    assertEquals(2.0, (Double) result.some(), 0.1);
}

@Test
public void option_test_failure() {
    Option result = FjRomanNumeralParser.divide(4.0, 0);
    assertEquals(Option.none(), result);

}
```

リスト 10 に示されているように、`Option` には none() または some() のいずれかが含まれます。これらは `Either` の left と right と似ていますが、`none()` や `some()` は正当な戻り値を持たない可能性のあるメソッドに特有です。

Functional Java での `Either` と `Option` はいずれもモナドです。つまり、計算を表す特殊なデータ構造であり、関数型言語では非常によく使用されています。次回の記事では、`Either` に関連するモナドの概念を探り、極端な例で Scala スタイルのパターン・マッチングを実現する方法を紹介します。

まとめ

新しいパラダイムを学ぶときには、問題を解決するのに慣れているあらゆる方法を見直さなければなりません。関数型プログラミングでは、さまざまなイディオムを使用してエラー条件をレポートします。これらのエラー条件のほとんどは、Java で再現することができますが、構文が複雑になることは認めざるを得ません。

次回の記事では、`Either` を使ってツリーを作成する方法を紹介します。

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。