関数型の考え方: 連結と合成、第 2 回

オブジェクト指向と関数型の構成概念の比較

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2011年 11月 04日

オブジェクト指向を構成する概念 (継承やポリモーフィズムなど) を使い慣れているプログラマーは、その欠点も、代わりとなる手段も目に入らなくなってしまいがちです。関数型プログラミングでは再利用を可能にするために、より汎用的な概念 (リスト変換や移植可能なコードなど) をベースにさまざまな構成概念を用います。連載「関数型の考え方」の今回の記事では、コードを再利用するメカニズムとして、継承による連結と合成とを比較し、命令型プログラミングと関数型プログラミングの重要な違いの 1 つを指摘します。

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。

前回の記事では、コードを再利用する手法として、オブジェクト指向の手法と関数型の手法を説明しました。オブジェクト指向の手法では、重複するメソッドを抽出し、`protected` フィールドと併せてこれらのメソッドをスーパークラスに移動します。一方、関数型の手法では、純粋な関数 (副次的な影響のない関数) を専用のクラスに抽出し、パラメーター値を指定して、そのクラスから関数を呼び出します。私は再利用のメカニズムを、継承による protected フィールドからメソッド・パラメーターに切り替えました。オブジェクト指向言語からなる機能 (継承など) には明らかな利点がありますが、想定外の副次的な悪影響を及ぼすこともあります。一部の読者から寄せられたコメントで鋭く指摘されているように、オブジェクト指向プログラミングで経験を積んだ多くの開発者はまさにこの理由から、継承によって状態を共有しないようにしています。しかし、オブジェクト指向のパラダイムが深くに染み付いている開発者にとって、継承に代わる手段はなかなか目に入りにくいものです。

今回の記事では再利用可能なコードを抽出する手段として、言語メカニズムによる連結と、移植可能なコードを使用した合成とを比較します。この比較によっても、コードの再利用に関する理念の重要な違いが明らかになります。記事では、まずは従来からの問題を取り上げます。その問題とは、継承の存在下で適切な `equals()` メソッドを作成する方法です。

`equals()` メソッドに関する問題

Joshua Bloch 氏の著書『Effective Java』に、適切な `equals()` メソッドと `hashCode()` メソッドの作成方法を取り上げたセクションがあります (「参考文献」を参照)。このセクションで説明しているように、これらのメソッドの作成を困難にしているのは、等価性の動作と継承との相互作用です。Java の `equals()` メソッドは、`Object.equals()` に求められる特性として Javadoc に明記されている、以下の特性を備えていなければなりません。

- 再帰性: すべての非ヌル参照の値 x に対して、x.equals(x) が true を返すこと
- 対称性: すべての非ヌル参照の値 x および y に対して、y.equals(x) が true を返す場合にのみ、x.equals(y) が true を返すこと
- 推移性: すべての非ヌル参照の値 x、y、z に対して、x.equals(y) が true を返し、かつ y.equals(z) が true を返す場合には、x.equals(z) が true を返すこと
- 一貫性: すべての非ヌル参照の値 x および y に対して、オブジェクトの等価性の比較に使用される情報が変更されないとするならば、x.equals(y) を何度呼び出しても常に true が返されるか、または常に false が返されること
- すべての非ヌル参照の値 x に対して、x.equals(null) が false を返すこと

Bloch 氏は例として、`Point` と `ColorPoint` という 2 つのクラスを作成し、この両方に対して正しく動作する `equals()` メソッドの作成を試みています。継承されたクラスの追加フィールドを無視しようとすると対称性が失われ、それを考慮しようとすると推移性が失われます。この問題に対し、Josh Bloch 氏は以下の厳しい結末を予想しています。

> 「equals メソッドの規約を守る一方で、インスタンス化可能なクラスを継承して、ある側面を追加する方法はありません。」

継承された可変フィールドについて心配する必要がなければ、等価性は遥かに簡単に実装できますが、継承のような連結メカニズムを追加すると、微妙な差異と落とし穴が生じます (結局、継承を維持しつつも、この問題を解決する方法は見つかりましたが、その方法には、依存メソッドをさらに追加するという代償を伴います。囲み記事「継承と `canEqual()`」を参照してください。

継承と `canEqual()`

『Programming Scala』の著者たちは、継承が存在していても等価性を有効にするメカニズムを提案しています (「参考文献」を参照)。Bloch 氏が取り上げている問題の核心は、親クラスがサブクラスを等価性の比較に関与させるかどうかを判断できるだけの「情報」を持っていないところにあります。この情報不足に対処するには、`canEqual()` メソッドを基底クラスに追加し、等価性の比較対象とする子クラスに対しては、このメソッドをオーバーライドします。こうすることにより、現行クラスが (`canEqual()` によって) 2 つのタイプを同一とみなすのが妥当で賢明なことであるかどうかを判断することができます。

このメカニズムによって問題は解決されますが、`canEqual()` によって親クラスと子クラスの間にさらに別の連結ポイントが追加されるという代償を伴います。

連載の前の 2 回の記事の冒頭で紹介した Michael Feathers 氏の言葉を思い出してください。

> 「オブジェクト指向プログラミングでは、可変の構成要素を「カプセル化する」ことによってコードを理解しやすくする一方、関数型プログラミングでは、可変の構成要素を「最小限にする」ことによってコードを理解しやすくします。」

`equals()` を実装することの難しさは、「可変の構成要素」に関する Feathers 氏の言葉を説明する良い例となります。継承は 1 つの連結メカニズムであり、可視性やメソッドのディスパッチなどについて明確に定義したルールによって 2 つのエンティティーを結び付けます。Java のような言語では、ポリモーフィズムも継承に関係します。これらの連結ポイントが、Java をオブジェクト指向言語にしている所以です。けれども、可変の構成要素を使用できるようにすると、言語レベルでは尚更のこと、そのことによって新たな結果を伴います。例えば、操縦士が 4 本の手足を使ってペダルやレバーを操作するヘリコプターは、操縦が難しいことで有名です。これらのペダルやレバーは連動することから、操縦士はそれぞれのペダルやレバーの操作が他へ及ぼす影響を上手く制御できるようにならなければなりません。言語の構成要素はヘリコプターのペダルやレバーのようなもので、他の構成要素に影響を与えずに、不用意に要素を追加 (または変更) することはできません。

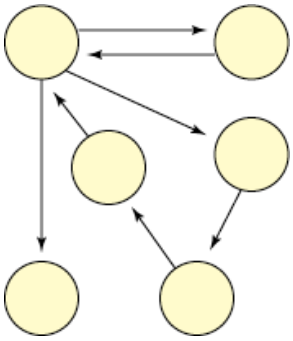
オブジェクト指向言語では継承があまりにも当たり前になっているので、継承が基本的には連結メカニズムであるという事実を、ほとんどの開発者が忘れています。思わぬ要素が壊れたり、機能しなかったりしても、開発者は問題を緩和するための (ときには不可解な) ルールを学んで作業を進めるだけです。けれども、これらの暗黙の連結ルールは、再利用、拡張性、そして等価性をどのように実現するかといった、コードの基本的な側面に対する考え方に影響を及ぼします。

Bloch 氏が等価性の問題に結論を出していなかったとしたら、『Effective Java』がそれほど取り上げられることはなかったでしょう。彼はこの問題を、本の最初の部分で述べた優れたアドバイスをもう一度紹介する好機として利用しました。そのアドバイスとは、継承よりも合成を優先することです。`equals()` 問題に対する彼のソリューションでは、連結ではなく、合成を使用しています。このソリューションは継承の使用を一貫して避けるため、`ColorPoint` は `Point` の 1 つのタイプになる代わりに、Point のインスタンスへの参照を持つようになっています。

合成と継承

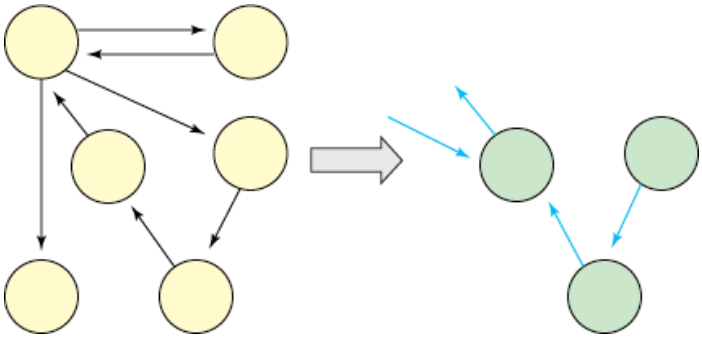
パラメーターの受け渡し、そして第一級関数という形での合成は、関数型プログラミング・ライブラリーでは再利用メカニズムとして頻繁に目にします。関数型言語で再利用を行うレベルはオブジェクト指向言語よりも粒度が粗く、再利用のために抽出されるのは、振る舞いがパラメーター化された共通コードです。一方、オブジェクト指向のシステムは、メッセージを他のオブジェクトに送信 (より具体的に言うと、メソッドを他のオブジェクトで実行) して通信するオブジェクトからなります。図 1 に、オブジェクト指向のシステムを表します。

図 1. オブジェクト指向のシステム



一連の有用なクラスとこれらのクラスが受け渡しするメッセージが見つかると、図式から該当するクラスを抽出して再利用します (図 2 を参照)。

図 2. グラフから有用な部分を抽出する



予想通り、ソフトウェア・エンジニアリングの世界で最も人気の高い本の 1 つは、まさに図 2 に示したようにパターンを抽出したものを集めたカタログである『オブジェクト指向における再利用のためのデザインパターン (原題: Design Patterns: Elements of Reusable Object-Oriented Software)』です。パターンによる再利用は広く浸透しており、他の多数の本でもこのように抽出したものを (異なる名前を付けて) カタログにしているほどです。デザイン・パターンには、名前が付けられているとともに、サンプルが用意されているため、デザイン・パターンを導入することで、ソフトウェア開発の世界には大きな恩恵がもたらされてきました。けれども、デザイン・パターンによる再利用は基本的に粒度が細かく、ソリューション同士は相容れません (例えば、Flyweight パターンと Memento パターン)。デザイン・パターンが解決する問題は、それぞれに極めて特有のものです。したがって、目の前の問題にぴったり適合するパターンが見つかることがよくあるため、これらのパターンは役に立ちますが、その一方であまりにも特定の問題に限定されることから、その有用性は範囲が限られます。

関数型プログラマーも再利用可能なコードを必要としますが、使用する構成概念は異なります。関数型プログラミングでは、構造間によく知られた関係 (連結) を作ろうとする代わりに、粒度の粗い再利用メカニズムを抽出しようと試みます。この抽出はその一部が圏論に基づいて行われます。圏論は、オブジェクトのタイプの間の関係 (射) を定義する数学理論の 1 つです (「参考文献」を参照)。大部分のアプリケーションでは要素のリストを用いて処理を行うため、関数型の手法ではリストの概念に加え、コンテキストに当てはめた移植可能なコードを中心とした再利用メカニズムを作成します。関数型言語では、パラメーターおよび戻り値として第一級関数 (他の言語構成体を使用できる場所であれば、どこででも使用できる関数) を使用します。この概念を図 3 に示します。

図 3. 粒度の粗いメカニズムと移植可能なコードによる再利用

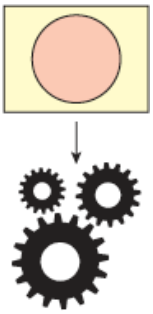


図 3 では、ギア・ボックスが、基本となるデータ構造を一般的な方法で扱う抽象化を表しており、黄色のボックスが、データをカプセル化した移植可能なコードを表しています。

共通の構成概念

この連載の 2 本目の記事では、Functional Java ライブラリー (「参考文献」を参照) を使用して数値分類子の例を作成しました。その際に 3 種類の構成概念を使用しましたが、これらの要素についての説明は省いたので、ここで説明します。

畳み込み

数値分類子で使用しているメソッドのうちの 1 つは、収集されたすべての約数を加算します。リスト 1 に、このメソッドを記載します。

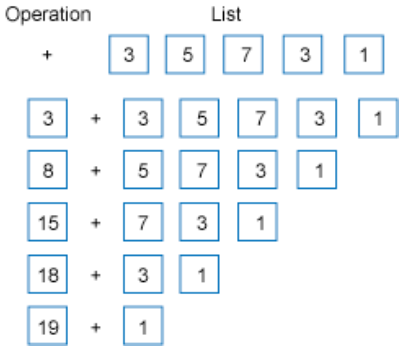
リスト 1. 関数型の数値分類子で使用している `sum()` メソッド

```
public int sum(List<Integer> factors) {
    return factors.foldLeft(fj.function.Integers.add, 0);
}
```

リスト 1 の 1 行のコードが加算演算を行う仕組みは、最初はわかりにくいでしょう。この例は、一般的なリスト変換手法における 1 つのタイプで、catamorphism と呼ばれます。この変換タイプは、ある形を別の形に変換します (「参考文献」を参照)。この例での畳み込み (fold) 操作は、リストの各要素をその隣の要素と結合していき、リスト全体で 1 つの結果に累積するという変換を意味します。foldLeft はリストにシード値が与えられた状態で開始し、リストの各要素を順番に結合することで、リストを左方向に畳み込んでいき、最終的に 1 つの結果を生成します。

図 4 に、畳み込み操作の一例を図解します。

図 4. 畳み込み操作



加算とは累積することなので、この場合には `foldLeft()` と `foldRight()` のどちらを行うかは問題ではありませんが、一部の演算 (減算、除算を含む) では順序が重要です。そのような場合に対処するために、対称の `foldRight()` メソッドがあります。

リスト 1 では、Functional Java ライブラリーが提供する `add` 列挙を使用しています。このライブラリーには、最も一般的な数学演算が含まれています。しかし、さらに詳細な基準が必要な場合はどうでしょうか。リスト 2 にその一例を記載します。

リスト 2. ユーザー指定の基準を使用する `foldLeft()`

```
static public int addOnlyOddNumbersIn(List<Integer> numbers) {
    return numbers.foldLeft(new F2<Integer, Integer, Integer>() {
        public Integer f(Integer i1, Integer i2) {
            return (!(i2 % 2 == 0)) ? i1 + i2 : i1;
        }
    }, 0);
}
```

Java には、ラムダ・ブロックという形での第一級関数 (「参考文献」を参照) がまだないため、Functional Javaでは Generics で対処せざるを得なくなります。畳み込み操作に適した構造は、組み込み F2 クラスにあります。`F2` クラスが作成するメソッドは、2 つの整数パラメーター (互いに畳み込まれる 2 つの値) と戻り型を受け入れるからです。リスト 2 の例では、2 番目の数値が奇数である場合にのみ、両方の数値の合計を返すことによって、奇数を合計します。それ以外の場合には、最初の数値だけを返します。

フィルタリング

リストでの一般的な操作には、フィルタリングもあります。つまり、ユーザー定義の基準に応じてリスト内の項目をフィルタリングすることによって、リストを絞り込むということです。図 5 に、フィルタリング操作を図解します。

図 5. リストのフィルタリング

フィルタリング操作によって、別のリスト (コレクション) が生成されます。そのリストは、フィルタリング基準によっては元のリストより小さくなる場合があります。数値分類子の例では、フィルタリングを使って数値の約数を判別しています (リスト 3 を参照)。

リスト 3. フィルタリングによる約数の判別

```
public boolean isFactor(int number, int potential_factor) {
    return number % potential_factor == 0;
}

public List<Integer> factorsOf(final int number) {
    return range(1, number + 1)
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
}
```

リスト 3 のコードは、1 から対象の数値までの数値の範囲を (`List` として) 作成した後、`filter()` メソッドを適用し、(リストの先頭に定義された) `isFactor()` メソッドを使用して対象の数値の約数ではない数値を削除します。

リスト 3 に示した機能は、クロージャーを使用できる言語では、遥かに簡潔に実現することができます。リスト 4 にフィルタリングを行う Groovy のコードを記載します。

リスト 4. Groovy でのフィルタリング操作

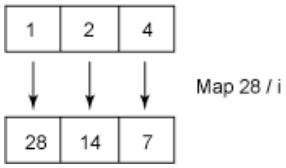
```
def isFactor(number, potential) {
  number % potential == 0;
}

def factorsOf(number) {
  (1..number).findAll { i -> isFactor(number, i) }
}
```

Groovy で `filter()` に相当する `findAll()` は、フィルタリング基準を指定するコード・ブロックを受け入れます。このメソッドの最後の行は、メソッドの戻り値です。この例の場合、戻り値は約数のリストとなります。

マップ操作

マップ操作は、コレクションの各要素に関数を適用することによって、新しいコレクションに変換します (図 6 を参照)。

図 6. 関数をコレクションに適用するマップ操作



数値分類子の例では、最適化した `factorsOf()` メソッドの中でマップ操作を使っています (リスト 5 を参照)。

リスト 5. Functional Java の `map()` を使用して最適化した約数検出メソッド

```
public List<Integer> factorsOfOptimized(final int number) {
    final List<Integer> factors = range(1, (int) round(sqrt(number) + 1))
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
    return factors.append(factors.map(new F<Integer, Integer>() {
        public Integer f(final Integer i) {
            return number / i;
        }
    }))
   .nub();
}
```

リスト 5 のコードは、最初に対象の数値の平方根までの約数を収集し、そのリストを factors 変数に格納します。次に、新しいコレクション (`factors` リストに `map()` 関数を適用して生成したコレ

クション) を `factors` に追加し、対称のリスト (平方根より大きい一致する約数のリスト) を生成するためのコードを適用します。そして最後の `nub()` メソッドで、リスト内に重複する要素がないことを確実にします。

例によって、Groovy のコード (リスト 6 を参照) はこれよりも遥かに単純明快になります。それは、柔軟なタイプとコード・ブロックがファーストクラス・オブジェクトであるからです。

リスト 6. Groovy の最適化された約数

```
def factorsOfOptimized(number) {
  def factors = (1..(Math.sqrt(number))).findAll { i -> isFactor(number, i) }
  factors + factors.collect({ i -> number / i})
}
```

メソッドの名前は異なりますが、リスト 6 のコードが実行するタスクはリスト 5 のコードと同じです。つまり、1 から対象の数値の平方根までの数値の範囲を取得し、それをフィルタリングによって約数に絞り込みます。そして、約数に絞り込んだリストの各値に、それらの値と対になる約数を生成する関数を適用してマップ操作をすることで、元の約数のリストに、対となる約数のリストを追加するという仕組みです。

関数型による完全数の検出

高階関数を使用できる Groovy では、数値が完全数であるかどうかを判別するという問題のすべてが、わずか数行のコードに集約されます (リスト 7 を参照)。

リスト 7. Groovy の完全数検出プログラム

```
def factorsOf(number) {
  (1..number).findAll { i -> isFactor(number, i) }
}

def isPerfect(number) {
    factorsOf(number).inject(0, {i, j -> i + j}) == 2 * number
}
```

もちろん、この例は数値分類に関する特殊なものなので、タイプの異なるさまざまなコードに対して一般化するのは難しいですが、私はいくつかのプロジェクトで、これらの抽象化をサポートする言語 (関数型言語であるか否かには関わらず) を使用するとコーディング・スタイルに顕著な違いが現れることに気付きました。この違いに初めて気付いたのは、Ruby on Rails プロジェクトにおいてです。Ruby にはクロージャー・ブロックを使用するリスト操作メソッドとして、これまでに説明したのと同様のメソッドがあり、`collect()`、`map()`、そして `inject()` が頻繁に出現することに驚かされました。これらのツールを使用するのにいったん慣れると、何度も繰り返し使用するようになるはずです。

まとめ

関数型プログラミングのような新しいパラダイムを習得する難しさの 1 つは、新しい構成概念について学び、それらの概念が問題のなかに潜在的なソリューションとして存在していることを「見つける」ことです。関数型プログラミングでは、抽象化はかなり少なくなるものの、それぞ

れの抽象化は汎用的です (第一級関数によって具体性が追加されます)。関数型プログラミングでは、パラメーターを渡して合成を行うことが非常に多いことから、可変の構成要素の間でのやりとりに関して学ばなければならないルールは数が限られます。そのため、ジョブは容易になります。

関数型プログラミングでは、高階関数によってカスタマイズできる、コードの汎用部分を抽象化するという方法でコードの再利用を実現します。この記事では、オブジェクト指向言語での継承による連結メカニズムがもたらす問題のいくつかに焦点を当て、クラスのグラフを抽出して再利用可能なコードにする一般的な方法を説明しました。これは、デザイン・パターンの領域です。次に、圏論をベースとした粒度の粗いメカニズムを使用することで、言語設計者が作成 (およびデバッグ) したコードを利用して問題を解決できる仕組みを紹介しました。このソリューションは、いずれの場合も簡潔な宣言型のソリューションです。これは、パラメーターおよび機能を合成して汎用的な振る舞いを作成するという、コードの再利用方法を説明する良い例となります。

次回の記事では、JVM 上で動作する動的言語である Groovy と JRuby が持つ関数型の機能をさらに深く探ります。

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。