関数型の考え方: 不変性

可変の部分を少なくして、Java コードをより関数型に近づける

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2011年 9月 02日

不変性は、関数型プログラミングの基本要素の 1 つです。連載「関数型の考え方」の今回の記事では、Java 言語における不変性のさまざまな側面を取り上げ、Java の不変クラスを作成する昔ながらの方法と新しい方法を説明します。さらに、Java 実装に伴う厄介な作業を大幅に取り除ける方法として、Groovy で不変クラスを作成する 2 つの方法を紹介します。そして最後に、この抽象化がどのような場合に適しているかを説明します。

このシリーズの他の記事を見る

「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。」
— 『Working with Legacy Code』の著者、Michael Feathers による Twitter への投稿

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。

今回の記事では、関数型プログラミングを構成する要素の 1 つ、不変性について説明します。不変オブジェクトは、作成された後にその状態を変更することができません。別の言い方をすれば、コンストラクターが唯一、オブジェクトの状態を変更する手段となります。不変オブジェクトを変更する場合には、そのオブジェクトを変更するのではなく、その変更後の値を持つオブジェクトを新しく作成し、そのオブジェクトを参照するしかありません (`String` は、Java 言語のコアに組み込まれた不変クラスの典型例です)。不変性は関数型プログラミングにとって重要な鍵となります。その理由は、不変性は可変の構成要素を最小限にすることで可変部分についての推論を容易にするという、関数型プログラミングの目標と合致するからです。

© Copyright IBM Corporation 2011

Java での不変クラスの実装

Java、Ruby、Perl、Groovy、C# などの最近のオブジェクト指向言語では、管理された方法で簡単に状態を変更できるようにする便利なメカニズムが構築されています。けれども、状態はコンピューティングにおいてあまりにも基本的な要素として使われるため、どこで管理から漏れるかはまったく予測することができません。例えば、オブジェクト指向言語で適切にマルチスレッド化されたハイパフォーマンスのコードを作成するには、多数の可変性メカニズムがその障害となります。Java は状態を操作するために最適化されていることから、不変性のメリットを活かすには、これらの可変性メカニズムのいくつかに対処しなければなりません。けれども、いくつかの落とし穴を避ける方法を学びさえすれば、Java でも簡単に不変クラスを作成できるようになります。

不変クラスを定義する

Java クラスを不変にするためには、以下のことが必要です。

- すべてのフィールドを `final` として定義すること。
  Java でフィールドを final として定義する場合、そのフィールドを宣言時に初期化するか、またはコンストラクターで初期化する必要があります。「宣言時にフィールドが初期化されていない」というエラーを IDE が出したとしても慌てないでください。コンストラクターに適切なコードを作成すれば、IDE はエラーでないことを認識します。
- クラスを `final` として定義して、オーバーライドできないようにすること。
  クラスがオーバーライド可能になっていると、そのクラスのメソッドの振る舞いもオーバーライドされる可能性があります。したがって、サブクラス化を許可しないことが最も安全な策です。これは、Java の `String` クラスで使用されるストラテジーです。
- 引数なしのコンストラクターを作成しないこと。
  不変オブジェクトを使用する場合には、そのオブジェクトに含まれる状態が何であろうとも、コンストラクターで状態を設定する必要があります。設定の対象となる状態がないとしたら、オブジェクトを使用する理由はあるでしょうか。ステートレスなクラスで static メソッドを使ったとしても不変オブジェクトと同じ効果があります。したがって、不変クラスには、引数なしのコンストラクターを決して使用しないでください。何らかの理由で引数なしのコンストラクターが必要となるフレームワークを使っている場合には、private として定義した引数なしのコンストラクター (リフレクションによって可視になります) を用意することで、その要件を満たせるかどうかを検討してください。
  引数なしのコンストラクターが存在しないということは、デフォルト・コンストラクターを要件とする JavaBean 標準に違反しますが、`setXXX` メソッドの仕組みにより、どのみち JavaBean を不変にすることはできません。
- 1 つ以上のコンストラクターを作成すること。
  引数なしのコンストラクターを作成していないとしたら、これが、オブジェクトに状態を追加する最後のチャンスです！
- コンストラクター以外に、オブジェクトを変更するメソッドを作成しないこと。
  通常の JavaBean に基づく `setXXX` メソッドを使わないようにするだけでなく、可変のオブジェクト参照を返さないように注意する必要もあります。オブジェクト参照が `final` であっても、その参照先を変更できないことにはなりません。したがって、`getXXX` メソッドから返されるあらゆるオブジェクト参照は、必ず (オブジェクトを複製することで) ディフェンシブ・コピー (defensive copy) を実行する必要があります。

「従来の」不変クラス

リスト 1 に、上記の要件を満たす不変クラスを記載します。

リスト 1. Javaでの不変 `Address` クラス

```
public final class Address {
    private final String name;
    private final List<String> streets;
    private final String city;
    private final String state;
    private final String zip;

    public Address(String name, List<String> streets,
                   String city, String state, String zip) {
        this.name = name;
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getName() {
        return name;
    }

    public List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }

    public String getCity() {
        return city;
    }

    public String getState() {
        return state;
    }

    public String getZip() {
        return zip;
    }
}
```

リスト 1 では、streets のリストのディフェンシブ・コピーを作成するために、`Collections.unmodifiableList()` メソッドを使用していることに注意してください。不変リストを作成するには、配列ではなく、必ずコレクションを使用してください。配列を変更できないようにコピーすることもできますが、そうすると、望ましくない副次効果を招きます。例えば、リスト 2 のコードを見てください。

リスト 2. コレクションの代わりに配列を使用した `Customer` クラス

```
public class Customer {
    public final String name;
    private final Address[] address;

    public Customer(String name, Address[] address) {
        this.name = name;
        this.address = address;
    }

    public Address[] getAddress() {
        return address.clone();
    }
}
```

リスト 2 のコードでの問題は、`getAddress()` メソッドを呼び出した結果として返される、複製された配列で何らかの操作を行おうとすると、明らかになります (リスト 3 を参照)。

リスト 3. 正しいながらも直観的ではない結果を示すテスト

```
public static List<String> streets(String... streets) {
    return asList(streets);
}

public static Address address(List<String> streets,
                              String city, String state, String zip) {
    return new Address(streets, city, state, zip);
}

@Test public void immutability_of_array_references_issue() {
    Address [] addresses = new Address[] {
        address(streets("201 E Washington Ave", "Ste 600"), "Chicago", "IL", "60601")};
    Customer c = new Customer("ACME", addresses);
    assertEquals(c.getAddress()[0].city, addresses[0].city);
    Address newAddress = new Address(
        streets("HackerzRulz Ln"), "Hackerville", "LA", "00000");
    // doesn't work, but fails invisibly
    c.getAddress()[0] = newAddress;

    // illustration that the above unable to change to Customer's address
    assertNotSame(c.getAddress()[0].city, newAddress.city);
    assertSame(c.getAddress()[0].city, addresses[0].city);
    assertEquals(c.getAddress()[0].city, addresses[0].city);
}
```

配列の複製を返すようにすれば、おおもとの配列を保護することになりますが、返される配列は通常の配列のように見えます。これはつまり、配列の内容を変更できるということです (配列を格納している変数が `final` であっても、その定義は配列の内容にではなく、配列の参照自体にしか適用されません)。`Collections.unmodifiableList()` (および、他のタイプの `Collections` でのメソッド群) を使用すれば、状態を変更するメソッドを 1 つも使用できないオブジェクト参照を受け取ることになります。

より簡潔な不変クラス

不変フィールドも private として定義しなければならないという意見をよく耳にしますが、私はこの意見に賛成しかねます。その根拠は、この以前からの思い込みを、独特ながらも明快な見解で、それが思い込みであることをはっきりさせている人物の意見を聞いたからです。Michael Fogus 氏による Clojure の作成者、Rich Hickey 氏へのインタビュー (「参考文献」を

参照) で、Hickey 氏は Clojure の多くのコア要素には、データを包み隠すカプセル化がないと語っています。状態をベースにした考え方に深くはまり込んでいる私にとって、Clojure のこの側面はいつも悩みの種となっていましたが、フィールドが不変であれば、フィールドの公開について心配する必要はないことに気付きました。私たちがカプセル化のために使っている保護対策の多くは、実際には状態の変更を防ぐためだけのものです。この 2 つの概念を切り分ければ、Java 実装はより簡潔になります。

リスト 4 に、`Address` クラスの別のバージョンを記載します。

リスト 4. 不変フィールドを public として定義した `Address` クラス

```
public final class Address {
    private final List<String> streets;
    public final String city;
    public final String state;
    public final String zip;

    public Address(List<String> streets, String city, String state, String zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public final List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }
}
```

不変フィールドに対して public `getXXX()` メソッドを宣言することにメリットがあるのは、おおもとの表現を隠したい場合だけですが、IDE のリファクタリング機能によって表現の変更をいとも簡単に見つけられる今の時代、それがメリットと呼べるかどうかはわかりません。フィールドを public として定義するだけでなく、不変にすることによって、誤って変更してしまうことを心配せずに、コード内で直接フィールドにアクセスすることができます。

コレクションを内部で変更する必要がまったくないのであれば、コンストラクター内に組み込まれているリストを `unmodifiableList` にキャストするという方法もあります。こうすると、`streets` フィールドを public にすることができるため、`getStreets()` メソッドを使用する必要がなくなります。次の例に示すように、Groovy では `getStreets()` のような保護 access メソッドを作成しながらも、フィールドとして現れるようにすることができます。

怒った猿たちの話を聞くと、不変の public フィールドを使うのは、初めは不自然のように思えますが、フィールドを差別化することにはメリットがあります。Java で不変タイプを扱うのに慣れていないとしたら、リスト 5 は新しいタイプのコードに見えることでしょう。

リスト 5. `Address` クラスのユニット・テスト

```
@Test (expected = UnsupportedOperationException.class)
public void address_access_to_fields_but_enforces_immutability() {
    Address a = new Address(
        streets("201 E Randolph St", "Ste 25"), "Chicago", "IL", "60601");
    assertEquals("Chicago", a.city);
    assertEquals("IL", a.state);
    assertEquals("60601", a.zip);
    assertEquals("201 E Randolph St", a.getStreets().get(0));
    assertEquals("Ste 25", a.getStreets().get(1));
    // compiler disallows
    //a.city = "New York";
    a.getStreets().clear();
}
```

怒った猿たち

Dave Thomas 氏から初めて聞いたこの話は、その後、私の著書『プロダクティブ・プログラマ』(「参考文献」を参照) でも取り上げました。しかし (かなり調査したにも関わらず) それが真実なのかどうかはわかりません。けれども、その真偽はどうでもよいことです。この話は要点を見事に突いています。

これは、1960年代に行動科学者たちが行った実験の話です。この実験では、脚立を置いて天井からバナナを吊るした部屋に 5 匹の猿を入れます。猿たちはすぐに、脚立を登ればバナナを食べられることに気付きました。続いて、猿が脚立に近づくたびに、部屋全体に氷のように冷たい水を浴びせかけます。すると、脚立に近づこうとする猿は一匹もいなくなりました。今度は、水を浴びせられた一匹の猿を、まだ実験には加わっていない新しい猿と入れ替えます。新しい猿が脚立に向かってまっしぐらに進んで行くと、他の猿たちがその猿を叩きのめしました。その猿はなぜ襲われたのかはわかりませんでしたが、脚立のそばには近づいてはいけないとすぐに学習しました。科学者たちは徐々に新しい猿に入れ替えていき、最終的には冷たい水を浴びせられたことのない猿だけのグループになりましたが、それでも脚立に近づく猿は攻撃されました。

その要点とは、ソフトウェア・プロジェクトでの慣例には、「これまで常にそうしてきた」という理由だけで存在する慣例が数多くあることです。

public として定義された不変フィールドにアクセスするのであれば、一連の `getXXX()` 呼び出しによってコードが読みにくくなることがなくなります。もう 1 つの注目すべき点として、コンパイラーを使用してプリミティブのいずれかに値を割り当てることはできません。状態を変更するメソッドを `street` コレクションで呼び出そうとしても、(テストの先頭でキャッチされる) `UnsupportedOperationException` を受け取るだけです。このスタイルのコードを使用すれば、これが不変クラスであることが一目瞭然となります。

欠点

簡素化された構文で考えられる欠点の 1 つは、この新しいイディオムを学ぶ際の苦労です。けれども、私は苦労するだけの価値はあると思います。なぜなら、スタイルには明らかな違いがあるため、クラスを作成するときに不変性について考えるようになり、不要なボイラープレート・コードが少なくなるためです。その一方、(公平に言って、不変性に直接対応するようには設計されていない) Java でのこのコーディング・スタイルには以下の欠点があります

- Glenn Vanderburg 氏が私に最大の欠点として指摘したように、このスタイルは、Bertrand Meyer 氏 (Eiffel プログラミング言語の作成者) が Uniform Access Principle (統一形式アクセスの原則) と呼ぶ原則に違反します。Uniform Access Principle とは、「モジュールが提供する

すべてのサービスは、ストレージに記憶される形で実装されようが、計算によって実装されようが、変わることのない統一された表記によって使用できるようにしなければならない」という原則です。別の言葉に置き換えると、フィールドへのアクセスは、それがフィールドであるか、値を返すメソッドであるかを露呈してはならないということです。その点から言うと、`Address` クラスの `getStreets()` メソッドは、他のフィールドと統一されていません。Java でこの問題を解決することはどうしてもできないので、他の JVM 言語で、不変性を実現するようにしなければなりません。
- リフレクションに大幅に依存するフレークワークは、デフォルト・コンストラクターを必要とするため、このイディオムでは機能しません。
- 既存の可変オブジェクトを変更する代わりに新しいオブジェクトを作成することから、システムでの多数の更新によって、ガーベッジ・コレクションが非効率的になる可能性があります。Clojure などの言語には、不変の参照によってガーベッジ・コレクションを効率化する機能が組み込まれており、こうした言語では、この機能がデフォルトとなっています。

Groovy での不変性

`Address` クラスの public 不変フィールドを Groovy で作成すると、実装はかなり簡潔になります (リスト 6 を参照)。

リスト 6. Groovy での不変 `Address` クラス

```
class Address {
    def public final List<String> streets;
    def public final city;
    def public final state;
    def public final zip;

    def Address(streets, city, state, zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    def getStreets() {
        Collections.unmodifiableList(streets);
    }
}
```

Groovy では例のごとく、必要とされるボイラープレート・コードは Java より少ないですが、それ以外にも Groovy にはメリットがあります。Groovy では、お馴染みの `get`/`set` 構文を使ってプロパティーを作成することができるため、オブジェクト参照に対してまさに保護されたプロパティーを作成することができます。例えば、リスト 7 のユニット・テストを見てください。

リスト 7. Groovy での統一されたアクセス方法を示すユニット・テスト

```
class AddressTest {
    @Test (expected = ReadOnlyPropertyException.class)
    void address_primitives_immutability() {
        Address a = new Address(
            ["201 E Randolph St", "25th Floor"], "Chicago", "IL", "60601")
        assertEquals "Chicago", a.city
        a.city = "New York"
    }

    @Test (expected=UnsupportedOperationException.class)
    void address_list_references() {
        Address a = new Address(
            ["201 E Randolph St", "25th Floor"], "Chicago", "IL", "60601")
        assertEquals "201 E Randolph St", a.streets[0]
        assertEquals "25th Floor", a.streets[1]
        a.streets[0] = "404 W Randoph St"
    }
}
```

いずれのケースにしても、不変性のコントラクトに対する違反が原因で例外がスローされると、テストが終了します。リスト 7 の `streets` プロパティーはプリミティブのように見えますが、実際にはその `getStreets()` メソッドによって保護されています。

Groovy の `@Immutable` アノテーション

この連載の根底にある信条の 1 つとして、関数型言語は、プログラマーの代わりに下位レベルの詳細を処理します。その好例は、Groovy のバージョン 1.7 に追加された `@Immutable` アノテーションです。このアノテーションは、リスト 6 のコーディング全体を実際的な意味の無いものにします。リスト 8 に、このアノテーションを使用した `Client` クラスを記載します。

リスト 8. 不変 `Client` クラス

```
@Immutable
class Client {
    String name, city, state, zip
    String[] streets
}
```

`@Immutable` アノテーションを使用することにより、このクラスには以下の特性が備わります。

- final クラスとなります。
- プロパティーには自動的に、get メソッドが同期された private 支援フィールドを持つことになります。
- プロパティーを更新しようとすると、必ず `ReadOnlyPropertyException` が返されます。
- Groovy が順序ベースのコンストラクターとマップ・ベースのコンストラクターの両方を作成します。
- コレクション・クラスが適切なラッパーにラップされて、配列 (および他の複製可能なオブジェクト) が複製されます。
- デフォルトの `equals`、`hashcode`、および `toString` メソッドが自動的に生成されます。

このアノテーションは、その役割に見合うだけの価値を提供するだけでなく、期待通りに機能します (リスト 9 を参照)。

リスト 9. 期待されるケースを適切に処理する `@Immutable` アノテーション

```
@Test (expected = ReadOnlyPropertyException)
void client_object_references_protected() {
    def c = new Client([streets: ["201 E Randolph St", "Ste 25"]])
    c.streets = new ArrayList();
}

@Test (expected = UnsupportedOperationException)
void client_reference_contents_protected() {
    def c = new Client ([streets: ["201 E Randolph St", "Ste 25"]])
    c.streets[0] = "525 Broadway St"
}

@Test
void equality() {
    def d = new Client(
        [name: "ACME", city:"Chicago", state:"IL",
         zip:"60601",
         streets: ["201 E Randolph St", "Ste 25"]])
    def c = new Client(
            [name: "ACME", city:"Chicago", state:"IL",
             zip:"60601",
             streets: ["201 E Randolph St", "Ste 25"]])
    assertEquals(c, d)
    assertEquals(c.hashCode(), d.hashCode())
    assertFalse(c.is(d))
}
```

オブジェクト参照を置き換えようとすると、`ReadOnlyPropertyException` が返されます。また、カプセル化されたオブジェクト参照の 1 つの参照先を変更しようとすると、`UnsupportedOperationException`が発生します。さらに、最後のテストに示されているように、このアノテーションは適切な `equals` メソッドと `hashcode` メソッドを作成します。つまり、これらのオブジェクトの内容は同じですが、同じ参照を指していないということです。

もちろん、Scala および Clojure も不変性をサポートおよび奨励し、そのための簡潔な構文を用意しています。その実装については、今後の記事で紹介します。

不変性がもたらすメリット

関数型プログラマーのように考える上で、不変性を取り込むことは高い優先順位に挙げられます。Java で不変オブジェクトを作成するには、多少の複雑な処理が事前に必要になりますが、この抽象化によってその後の処理が単純化されることで、その努力はたちまち報われます。

不変クラスは、Java につきものの厄介な問題の多くを取り去ります。関数型の考え方に切り替えるメリットの 1 つは、コードのなかで変更が正常に行われていることを確認するためのテストが存在している点です。別の言い方をすると、テストの本当の目的は、状態の変更を検証することです。状態の変更が多ければ多いほど、適切に変更が行われているかどうかを確かめるためのテストが数多く必要になってきます。状態の変更を厳しく制限して、変更が行われる箇所を分離すれば、エラーが発生する余地は遥かに小さくなり、テストする箇所も少なくなります。不変クラスの場合、変更はオブジェクトの作成時にしか行われないため、ユニット・テストの作成が極めて容易になります。コンストラクターをコピーする必要も、`clone()` メソッドを実装するための事細かな大変さを心配する必要もありません。`Map` または `Set` のいずれかでキーとして使用するには、不変オブジェクトが有力候補になります。Java での辞書コレクションのキーは、キーとして使用されている間は値を変更できないため、不変オブジェクトは理想的なキーです。

不変オブジェクトは、本来スレッドセーフでもあり、同期の問題とは無縁です。不変クラスは不明の状態、あるいは望ましくない状態になることもありません。その場合には、例外がスローされます。すべての初期化は作成時に行われます。これは Java で行われるアトミックな処理であるため、あらゆる例外はオブジェクト・インスタンスを取得する前に発生します。Joshua Bloch 氏はこれを、「failure atomicity (失敗のアトミック性)」と呼んでいます。可変性に基づく成功または失敗は、オブジェクトがいったん作成されれば、永遠に解決されます (「参考文献」を参照)。

最後に、不変クラスの最も優れた特徴の 1 つとして挙げられるのは、「composition (合成)」の抽象化にぴったり適合することです。次回の記事では、合成について調べるところから初め、関数型の考え方で合成がこれほどまでに重要となる理由を探ります。

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。

© Copyright IBM Corporation 2011
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)