Javaの理論と実践: 可変性か、不変性か?

不変オブジェクトでプログラミングはかなり簡素化できる

Brian Goetz
Principal Consultant
Quiotix

2002年 2月 18日

不変オブジェクトは、同期の必要性が少ないため、データの破損を気にすることなくオブジェクト参照を共有およびキャッシュできるなど、多くの使いやすい特性を備えています。不変性はすべてのクラスにおいて有効であるとは言えませんが、ほとんどのプログラムには、不変であることでメリットを得るクラスが少なくともいくつか存在します。今月のJavaの理論と実践では、不変性のメリットと、不変クラス作成時のガイドラインについて、Brian Goetz氏が説明します。

このシリーズの他の記事を見る

不変オブジェクトとは、インスタンス化された後、外部から見た状態が変わらないオブジェクトのことです。不変オブジェクトには、Javaクラス・ライブラリーの`String`、`Integer`、`BigDecimal`クラスなどがあります。これらのオブジェクトは、そのオブジェクトの存続期間にわたって変更されない単一の値を表します。

不変性のメリット

不変クラスを適切に使用することでプログラミングを大幅に簡素化できます。不変クラスは1つの状態しか持たないため、オブジェクトが正しく作成されてさえいれば、矛盾した状態に陥ることはありません。不変オブジェクトへの参照は、コピーやクローン化を行うことなく、自由に共有およびキャッシュできます。不変オブジェクトのフィールドやメソッドの結果をキャッシュしても、それらが無効になったり、そのオブジェクトの他の状態と矛盾する心配はありません。不変クラスは、通常、最も優れたマップ・キーになります。また不変クラスは生来スレッド・セーフであるため、スレッド間でクラスへのアクセスで同期をとる必要もありません。

自由なキャッシュ

不変オブジェクトでは値が変化する危険性がないため、不変オブジェクトへの参照をキャッシュする場合、その参照がその後も必ず同じ値を指すことを前提に自由にキャッシュを実行できます。同様に、不変オブジェクトのプロパティーも変化しないため、不変オブジェクトのフィールドやメソッドの結果もキャッシュできます。

オブジェクトが可変の場合は、そのオブジェクトへの参照を保存するときに、いくつか気をつけなければならない点があります。リスト1のコードを見てください。ここではスケジューラーで実行する2つのタスクをキューに入れています。このコードは、最初のタスクを今実行し、2番目のタスクを次の日に実行することを目的としています。

リスト1. 可変であるDateオブジェクトの潜在的な問題点

```
Date d = new Date();
Scheduler.scheduleTask(task1, d);
d.setTime(d.getTime() + ONE_DAY);
scheduler.scheduleTask(task2, d);
```

`Date` は可変であるため、`scheduleTask` メソッドでは、十分な注意を払って (`clone()` などを使用して) 防御的に日付パラメーターを内部データ構造にコピーする必要があります。そうでないと、`task1` と`task2` は両方とも明日実行される可能性があり、それでは予定の動作と異なります。ひどい場合には、タスク・スケジューラーが使用する内部データ構造が破損する可能性もあります。`scheduleTask()` のようなメソッドを記述する場合には、日付パラメーターを防御的にコピーすることを忘れがちです。これを忘れると、しばらくの間は表面化しないものの、いざ起こってみると追跡に長い時間のかかるわかりにくいバグを作ることになります。`Date` クラスが不変であれば、この種のバグが発生することはないでしょう。

生来のスレッド・セーフティー

スレッド・セーフティーに関する問題のほとんどは、複数のスレッドが1つのオブジェクトの状態を同時に変更しようとする場合 (書き込み - 書き込み競合) か、1つのスレッドがオブジェクトの状態にアクセスしようとしているときに別のスレッドがそれを修正している場合 (読み取り - 書き込み競合) に発生します。このような競合を防ぐには、共有オブジェクトへのアクセスを同期化し、矛盾がある状態のオブジェクトに他のスレッドがアクセスできないようにします。これを確実に実行するのは簡単なことではありません。プログラムが適切に拡張されるよう膨大なドキュメントを残す必要があると同時に、パフォーマンスにマイナスの影響が出る場合があります。不変オブジェクトが正しく (つまり、オブジェクト参照がコンストラクター外部に渡らないように) 作成されてさえいれば、そのオブジェクトの状態は変更されないため "書き込み - 書き込み競合" も "読込み - 書き込み競合" も発生せず、したがってアクセスの際に同期を取る必要もなくなります。

同期化を必要とせずにスレッド間で自由に不変オブジェクトへの参照を共有できれば、並行プログラムの作成プロセスを大幅に簡素化でき、プログラムの実行時に発生する可能性のある並行に関連した問題も少なくなります。

無作法なコードに対しての安全性

オブジェクトを引数として受け取るメソッドでは、明示的にドキュメント化されているか、そのオブジェクトの所有権が事実上明らかである場合を除いては、オブジェクトの状態を変更すべきではありません。通常のメソッドにオブジェクトを渡す場合、一般的には、そのオブジェクトが変更されて返されることを想定しません。ただし、可変オブジェクトの場合、これはただの賭にすぎません。たとえば、`java.awt.Point` を`Component.setLocation()` のようなメソッドに渡せば、`setLocation` は自由にその`Point` の位置を変更したり、その点への参照を保存した後に別のメソッドで変更することができます。(もちろん`Component` ではこのような無作法なことは起こりませんが、すべてのクラスがこのように礼儀正しいとは限りません。) これで`Point` の状態は知ら

ない間に変更され、危険な結果を招く可能性を持つことになります。しかも、私たちはまだ、実際は別の場所にあるこの点が正しい場所にあると信じているのです。しかし`Point`が不変であれば、このようにありがたくないコードでも、プログラムの状態をこれほど紛らわしく危険な方法で変更することはありません。

優れたキー

不変オブジェクトは、`HashMap`や`HashSet`の最も優れたキーとなります。可変オブジェクトの中には、その状態によって`hashCode()`の値を変更するものがあります (リスト2の`StringHolder`クラスなど)。このような可変オブジェクトを`HashSet`のキーにすると、そのオブジェクトの状態が変化した場合に`HashSet`の実装が混乱します。つまり、そのセットを列挙したときにはオブジェクトが存在するように見えるのに、`contains()`を使用した照会では存在しないように見える可能性があります。これによってプログラムの動作が混乱することは言うまでもありません。リスト2はそれを示したコードです。実行すると、「false」、「1」、「moo」と表示されます。

リスト2. キーとしての使用に適さない可変のStringHolderクラス

```
public class StringHolder {
    private String string;
    public StringHolder(String s) {
        this.string = s;
    }
    public String getString() {
        return string;
    }
    public void setString(String string) {
        this.string = string;
    }
    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o == null || !(o instanceof StringHolder))
            return false;
        else {
            final StringHolder other = (StringHolder) o;
            if (string == null)
                return (other.string == null);
            else
                return string.equals(other.string);
        }
    }
    public int hashCode() {
        return (string != null ? string.hashCode() : 0);
    }
    public String toString() {
        return string;
    }
    ...
    StringHolder sh = new StringHolder("blert");
    HashSet h = new HashSet();
    h.add(sh);
    sh.setString("moo");
    System.out.println(h.contains(sh));
    System.out.println(h.size());
    System.out.println(h.iterator().next());
}
```

いつ不変クラスを使用するか

不変クラスは、数値、列挙型、色など、抽象データ型の表現に非常に適しています。Javaクラス・ライブラリーの基本的な数値型である`Integer`、`Long`、および`Float`や、その他の標準的な数値型である`BigInteger`や`BigDecimal`も不変クラスです。複雑な数値や任意の精度を持つ有理数は、不変クラスの候補になります。アプリケーションによっては、ベクトルやマトリックスなどの多くの計数値を持つ抽象型でも、不変クラスとして実装する候補になる場合があります。

Flyweightパターン

不変性はFlyweightパターンを可能にします。Flyweightパターンとは、共有によってオブジェクトの使用を容易にすることで、小さくて粒度の高い多数のオブジェクトを効率的に生成する方法です。たとえば、ワープロ文書の各文字やイメージの各ピクセル1つ1つをオブジェクトで表現するとします。しかし、このような単純な考えをそのまま実装すれば、きわめて大量のメモリーと、メモリー管理上のオーバーヘッドに法外なコストがかかるでしょう。Flyweightパターンは、ファクトリー・メソッドを使用して細粒度の高い不変オブジェクトへの参照を作成し、共有を使用して、たとえば文字「a」に対応するオブジェクトのインスタンスを1つしか持たないようにすることでオブジェクト数を削減します。Flyweightパターンの詳細については、『Design Patterns』(Gamma他) を参照してください(参考文献を参照)。

Javaクラス・ライブラリーにおける不変クラスのもう1つの代表例は`java.awt.Color`です。色は一般的に、何らかの色表現 (RGB、HSB、CMYKなど) の数値が規則的に並んだセットとして表現されます。しかし、色は、個別にアドレス指定が可能な値が規則的に並んだセットであると考えるよりも、色空間に存在するそれぞれに区別された値として考える方がわかりやすくなります。したがって、`Color`を不変クラスとして実装することは、道理にかなっています。

それでは、複数のプリミティブな値のコンテナーである、点、ベクトル、マトリックス、RGBカラーなどのオブジェクトは、可変オブジェクトとして表現すべきでしょうか。答えは、次のことを考慮した結果によって異なります。対象オブジェクトはどのように使用されるのか。それらは主に多次元の値 (ピクセルの色など) を表現するために使用されているのか、単にそれ以外のオブジェクトの関連するプロパティーのコレクション (ウィンドウの高さと幅など) のコンテナーとして使用されているのか。それらのプロパティーはどのような頻度で変更されるのか。プロパティーが変更される場合、個々のコンポーネントの値はそれ自身アプリケーション内で意味を持つのか。

イベントも、不変クラスとしての実装を検討すべき候補です。イベントは短命で、多くの場合、作成元のスレッドとは異なるスレッドで使用されます。そのためイベントを不変にすると、デメリットよりもメリットの方が多くなります。ほとんどのAWTイベント・クラスは、厳密には不変クラスとして実装されていませんが、少し変更するだけで不変にすることができます。同様に、コンポーネント間の通信に何らかの形態のメッセージングを使用しているシステムでは、メッセージ・オブジェクトを不変にすることも賢明な判断です。

不変クラスの作成ガイドライン

不変クラスの作成は簡単です。以下の条件をすべて満たしていれば、そのクラスは不変クラスになります。

- すべてのフィールドがfinalである

- クラスがfinalとして宣言されている
- 作成時に`this`参照がコンストラクター外部に渡されない
- 配列、コレクションなどの可変オブジェクトや、`Date`などの可変クラスへの参照を含むフィールドが以下の条件を満たす
  - privateである
  - 返されないか、呼び出し側に公開されない
  - 参照の対象となるオブジェクトへの唯一の参照である
  - 参照するオブジェクトの状態を、そのオブジェクトが作成された後で変更しない

条件の最後のグループは複雑に思えるかもしれません。しかし大切なのは、配列やその他の可変オブジェクトへの参照を保存する場合、その可変オブジェクトにアクセスするクラスが現在のクラスだけになるようにする (そうでないと、他の誰かによってその可変オブジェクトの状態が変更される可能性がある) とともに、作成後にはその可変オブジェクトの状態を変更しないようにすることです。Java言語には、final配列 の要素が変更されないように強制する方法がないため、不変オブジェクトに配列への参照を保存するには、このような複雑な条件を満たす必要があります。配列の参照やその他の可変フィールドが、コンストラクターに渡された引数で初期化されている場合には、呼び出し側から渡される引数を安全策をとってコピーする必要があります。そうしない限り、その配列への排他的なアクセスは保証されません。このようにしなければ、呼び出し側がコンストラクターを呼び出した後でその配列の状態を変更することができるからです。リスト3は、呼び出し側から渡される配列を保存する、不変オブジェクトの正しい作成方法と間違った作成方法を示しています。

リスト3. 不変オブジェクトのコードの正しい作成方法と間違った作成方法

```
class ImmutableArrayHolder {
  private final int[] theArray;
  // Right way to write a constructor -- copy the array
  public ImmutableArrayHolder(int[] anArray) {
    this.theArray = (int[]) anArray.clone();
  }
  // Wrong way to write a constructor -- copy the reference
  // The caller could change the array after the call to the constructor
  public ImmutableArrayHolder(int[] anArray) {
    this.theArray = anArray;
  }
  // Right way to write an accessor -- don't expose the array reference
  public int getArrayLength() { return theArray.length }
  public int getArray(int n)  { return theArray[n]; }
  // Right way to write an accessor -- use clone()
  public int[] getArray()       { return (int[]) theArray.clone(); }
  // Wrong way to write an accessor -- expose the array reference
  // A caller could get the array reference and then change the contents
  public int[] getArray()       { return theArray }
}
```

もう少し手間をかければ、final以外のフィールドを使用する不変クラスを作成することもできます (たとえば、標準の`String`の実装では、`hashCode`値の遅延計算が使用されています)。その方が、厳格にfinalだけで作成されたクラスよりもパフォーマンスが向上することがあります。作成するクラスが数値型や色などの抽象型を表す場合には、`hashCode()`や`equals()`メソッドを使用して、そのオブジェクトを`HashMap`または`HashSet`のキーとして使用することができます。スレッドの安全性を維持するために、`this`参照がコンストラクターの外部に渡らないようにすることも重要です。

変更頻度の低いデータ

データの中には、プログラムの存続期間中ずっと一定のものもあれば、頻繁に変更されるものもあります。定数データは、明らかに不変性を持たせるべき候補であり、複雑で頻繁に変更される状態を持つオブジェクトは、一般に不変クラスとしての実装には適しません。では、頻繁というほどではなく、ときどき変更されるデータについてはどうでしょうか。ときどき変更されるデータが、手軽さとスレッド・セーフティーという不変性のメリットを享受する方法はあるのでしょうか。

`util.concurrent` パッケージの`CopyOnWriteArrayList` クラスは、ときどき変更を許可しながら不変性の優れた点を活用している良い例です。このクラスは、ユーザー・インターフェース・コンポーネントなどに代表される、イベント・リスナーをサポートするクラスでの使用に適しています。イベント・リスナーのリストは変更可能ですが、イベントの生成と比べればずっと少ない頻度でしか変更されません。

`CopyOnWriteArrayList` は、リストが変更されると、その基になる配列を変更する代わりに新しい配列を作成して古い配列を破棄しますが、それ以外の点では`ArrayList` クラスにきわめて近い動作をします。つまり、基になる配列への参照を内部的に保持する反復子を呼び出し側が取得しても、その反復子によって参照される配列は実際には変更不可であるため、同期をとったり、同時変更のリスクを気にすることなくトラバースできます。これにより、トラバースの前にリストのクローンを作成したり、トラバース中にリストを同期化するなど、不便でエラーを招きやすい、確実にパフォーマンスに悪影響を及ぼす操作を排除できます。特定の状況ではよくあることですが、挿入や削除に比べてずっとトラバースの頻度が高い場合には、`CopyOnWriteArrayList` を使用する方がパフォーマンスが向上してアクセスも便利になります。

まとめ

不変オブジェクトは、可変オブジェクトに比べてずっと操作が簡単です。不変オブジェクトは1つの状態しか持たず、常に整合性を保っているため、本質的にスレッド・セーフであり、自由に共有できます。検出が難しく、しかもよく起こるプログラミング・エラーには、不変オブジェクトの使用によって完全に排除できるものがたくさんあります。たとえば、スレッド間でアクセスを同期する際の失敗や、配列またはオブジェクトへの参照を保存する前のクローン化の失敗などです。クラスを作成するときには、常に、そのクラスを不変クラスとして有効に実装できるかどうかを自問自答することが重要です。その答えがイエスであることがいかに多いかに驚くことでしょう。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。