Javaの理論と実践: ハッシュの徹底

hashCode() とequals() の効果的で正確な定義

Brian Goetz
Principal Consultant
Quiotix

2003年 5月 27日

すべてのJavaオブジェクトには、`hashCode()` メソッドと`equals()`メソッドがあります。多くのクラスでは、オブジェクト・インスタンス間により高い意味的な互換性を提供するために、これらのメソッドのデフォルトの実装をオーバーライドしています。Javaの理論と実践の今回の記事では、Java開発者であるBrian Goetz氏が、Javaクラスの作成時に`hashCode()` と`equals()` を効果的かつ適切に定義するために従うべき規則とガイドラインについて説明します。

このシリーズの他の記事を見る

Java言語は、連想配列 (索引に任意のオブジェクトを使用できる配列) を直接サポートしていませんが、ルートの`Object` クラスに`hashCode()` メソッドが存在することから、明らかに`HashMap` (およびそれ以前の`Hashtable`) をあらゆる場面で使用することが予想されていたと考えられます。理想的な状況の下では、ハッシュに基づくコンテナーは効率的なデータの挿入と取得の両方を提供します。オブジェクト・モデルで直接ハッシュをサポートすることにより、ハッシュに基づくコンテナーの開発と使用は容易になります。

等価性の定義

`Object` クラスには、オブジェクトの同一性を判断するために、`equals()` と`hashCode()` の2つのメソッドがあります。この2つのメソッドの間には崩してはならない重要な関係があるため、一般にこれらのメソッドのいずれか一方をオーバーライドする場合には、両方をオーバーライドする必要があります。特に、2つのオブジェクトが`equals()` メソッドにより等価である場合には、その2つのオブジェクトは`hashCode()` でも同じ値を持たなければなりません (ただし、一般にこの逆の場合はそうとは限りません)。

特定のクラスで`equals()` が持つ意味は、そのクラスの実装者に任されています。特定のクラスの`equals()` の意味を定義することは、クラスの設計作業の一部ということになります。`Object` で提供されている次のデフォルトの実装は、単なる参照の等価性です。

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

このデフォルトの実装では、2つの参照がまったく同じオブジェクトを指している場合にのみ、これらの参照が等価だとみなされます。同様に、Object で提供されているhashCode() のデフォルトの実装は、対象オブジェクトのメモリー・アドレスを整数値にマッピングすることを基にしています。一部のアーキテクチャーでは、アドレス空間がint の値の範囲より大きいために、2つの異なるオブジェクトが同じhashCode() の値を持つ可能性があります。hashCode() をオーバーライドしても、System.identityHashCode() メソッドを使用すればこのデフォルトの値にアクセスすることができます。

equals() のオーバーライド-- 単純な例

同一性に基づいたequals() とhashCode() の実装は道理にかなったデフォルトですが、一部のクラスでは等価性の定義を多少緩やかなものにする方が望ましい場合があります。たとえば、Integer クラスでは、equals() はおおよそ次のように定義されています。

```
public boolean equals(Object obj) {
  return (obj instanceof Integer && intValue() == ((Integer) obj).intValue());
}
```

この定義では、2つのInteger オブジェクトは、同じ整数値を含んでいる場合にのみ等価になります。Integer をHashMap のキーとして使用することが実用向きであるのは、この事実と、Integer が不変オブジェクトであるという事実によります。値に基づくこの等価性評価の方法は、Integer、Float、Character、Boolean、String など、Javaクラス・ライブラリーのすべてのプリミティブ・ラッパー・クラスで使用されています (String オブジェクトの場合は、2つのStringオブジェクトが同じ文字の並びからなっている場合に等価になります)。これらのクラスは、変更不可であるとともに、hashCode() とequals() を実用に適した方法で実装しているため、すべてが優れたハッシュ・キーになります。

equals() とhashCode() をオーバーライドする理由

Integer でequals() とhashCode() がオーバーライドされていなかったとしたら、どうなるでしょうか。Integer をHashMap などのハッシュに基づくコレクションのキーとして使用しなければ、何も起こりません。しかし、もしそのようなInteger オブジェクトをHashMap でキーに使用したとすると、put() の呼び出しで使用したInteger のインスタンスとまったく同じインスタンスをget() の呼び出しで使用しない限り、そのキーに関連付けられた値を確実に取得することはできなくなります。これは、ある1つの整数値に対応するInteger オブジェクトのインスタンスをプログラム全体で1つしか使用しないようにする必要があるということです。言うまでもなく、このような方法は不便で、エラーが発生しやすくなります。

Object のインターフェースに関する取り決めでは、equals() の結果2つのオブジェクトが等価であるならば、その2つのオブジェクトは同じhashCode() の値を持たなければならないことになっています。ルートのオブジェクト・クラスでは、hashCode() の識別機能が完全にequals() に包含されているにもかかわらず、なぜhashCode() が必要なのでしょうか。hashCode() メソッドは、純粋に効率性のためだけに存在しています。Javaプラットフォームの設計者たちは、一般的なJavaアプリケーションにおけるハッシュに基づくコレクション・クラス (Hashtable、HashMap、HashSet など) の重要性を予見していましたし、equals() で多くのオブジェクトを比較するとコンピューター的にコストが高くなる可能性があります。すべてのJavaオブジェクトでhashCode() をサポートするようにすれば、ハッシュに基づくコレクションを使用して効率的に保存と取得を実行できます。

equals() とhashCode() の実装の要件

`equals()` と`hashCode()` の動作にはいくつかの制限が課されており、それらは`Object` のドキュメンテーションに列挙されています。特に、`equals()` メソッドは、次の特性を備えている必要があります。

- 対称性: 2つの参照、`a` および`b` で、`a.equals(b)` が真となるのは`b.equals(a)` が真の場合だけである。
- 再帰性: すべての非ヌル参照で`a.equals(a)` が真となる。
- 推移性:`a.equals(b)` と`b.equals(c)` がともに真の場合、`a.equals(c)` も真となる。
- `hashCode()` との一貫性: 2つの等価なオブジェクトは同じ`hashCode()` の値を持たなければならない。

`Object` の仕様では、`equals()` と`hashCode()` が一貫性を持つ、すなわち、「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、2つのメソッドの呼び出し結果は同じになる、というあいまいなガイドラインが提供されています。これは、「計算結果は元の値が変更されない限り変わることはない」と言っているように聞こえます。このあいまいな記述は、一般には、等価性とハッシュ値の計算はオブジェクトの状態を決定付ける機能に過ぎないということを意味していると解釈されています。

等価性の意味

Objectクラスの仕様で`equals()` と`hashCode()` に課せられている要件には、非常に簡単に適合することができます。しかし、`equals()` をオーバーライドするかどうかや、どのようにオーバーライドするかを決定するには、もう少し判断が必要です。`Integer` をはじめとする単純な不変値のクラス (実際にはほとんどすべての不変クラス) では、選択はきわめて明白で、等価性は根底にあるオブジェクトの状態の等価性に基づいて判断されるべきです。`Integer` の場合、オブジェクトの唯一の状態はその根底にある整数値です。

しかし、可変オブジェクトの場合、答えはそれほど明確であるとは限りません。`equals()` と`hashCode()` は、オブジェクトの同一性 (デフォルトの実装のような) とオブジェクトの状態 (IntegerやStringのような) のどちらに基づくべきでしょうか。簡単に答えを出すことはできません。答えは、クラスをどのように使用するのかによって変わってきます。`List` や`Map` などのコンテナーでは、どちらのオーバーライド方法をとっても筋の通った説明を付けることができるでしょう。コンテナー・クラスをはじめとするJavaクラス・ライブラリーのほとんどのクラスでは、オブジェクトの状態に基づいて`equals()` と`hashCode()` を実装するのが無難です。

オブジェクトの`hashCode()` の値がオブジェクトの状態に基づいて変わり得る場合、そのようなオブジェクトをハッシュに基づくコレクションのハッシュ・キーとして使用するときには、オブジェクトの状態の変更を許可しないように注意しなければなりません。すべてのハッシュに基づくコレクションでは、コレクション内でオブジェクトがキーとして使用されている間は、オブジェクトのハッシュ値が変更されないことが前提になっています。もしコレクション内でキーのハッシュ・コードが変更されるようなことがあれば、予測のできない混乱した結果を招くことになります。`List` のような可変オブジェクトを`HashMap` のキーに使用することは一般的な慣例ではないため、通常このようなことが実際の問題になることはありません。

状態に基づいてequals() およびhashCode() を定義する簡単な可変クラスの例は、Point です。2つのPoint オブジェクトは、同じ(x, y) 座標を参照していれば等価であり、Point のハッシュ値はx およびy 座標値のIEEE 754のビット表現に基づいています。

より複雑なクラスでは、equals() とhashCode() の動作がスーパークラスまたはインターフェースの仕様の強制を受ける場合もあります。たとえば、List インターフェースの場合、List オブジェクトが他のオブジェクトと等価になるのは、もう一方のオブジェクトもList オブジェクトで、自身と同じ要素 (各要素のObject.equals() で判別) が同じ順序で含まれている場合だけに限られています。hashCode() の要件はさらに具体的で、リストのhashCode() の値は、次の計算に適合するものでなければならないと定義されています。

```
 hashCode = 1;
 Iterator i = list.iterator();
 while (i.hasNext()) {
     Object obj = i.next();
     hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
 }
```

ハッシュ値がリストの内容に依存するだけでなく、個々の要素のハッシュ値を結び付けるために特定のアルゴリズムが指定されています (String クラスでも、String のハッシュ値の計算に同様のアルゴリズムを使用するように指定されています)。

独自のeuals() およびhashCode() メソッドの作成

デフォルトのequals() メソッドのオーバーライドはとても簡単ですが、既にオーバーライド済みのequals() メソッドを対称性と推移性のどちらの要件にも抵触することなくオーバーライドするには、きわめて細かい注意が必要です。equals() をオーバーライドする場合には、オーバーライド後のクラスを継承する必要のある他のユーザーが正しく継承を行えるように、equals() に必ずJavadocのコメントを付けておくべきです。

単純な例として、次のクラスについて考えてみましょう。

```
 class A {
   final B someNonNullField;
   C someOtherField;
   int someNonStateField;
 }
```

このクラスのequals() メソッドは、どのように記述すれば良いでしょうか。次の方法は、多くの状況に適しています。

```
 public boolean equals(Object other) {
   // Not strictly necessary, but often a good optimization
   if (this == other)
     return true;
   if (!(other instanceof A))
     return false;
   A otherA = (A) other;
   return (someNonNullField.equals(otherA.someNonNullField))
       && ((someOtherField == null) ? otherA.someOtherField == null :
someOtherField.equals(otherA.someOtherField)));
 }
```

equals() の定義が終わったところで、今度はこれと矛盾しないようにhashCode() を定義する必要があります。それほど便利とは言えませんが、矛盾のない方法として、次のようにhashCode() を定義することができます。

```
public int hashCode() { return 0; }
```

この方法を使用すると、HashMap で大量のエントリーを処理する際にパフォーマンスが著しく悪化しますが、前述の仕様には確実に適合しています。A のhashCode() のより賢明な実装方法は、次のとおりです。

```
  public int hashCode() { int hash = 1;
    hash = hash * 31 + someNonNullField.hashCode();
    hash = hash * 31 + (someOtherField == null ? 0 : someOtherField.hashCode());
    return hash;
  }
```

上記の2つの実装では、ともに計算や評価の一部がクラスの状態フィールドのequals() またはhashCode() メソッドに委譲されています。クラスによっては、計算や評価の一部をスーパークラスのequals() またはhashCode() 関数に委譲する必要がある場合もあります。プリミティブ・フィールドについては、Float.floatToIntBits のように、関連付けられたラッパー・クラス内にヘルパー関数があり、そのヘルパー関数を使用して容易にハッシュ値を作成することができます。

equals() メソッドの作成には、思わぬ危険があります。一般に、それ自体がequals() をオーバーライドしているインスタンス化可能なクラスを継承するときに、equals() を簡単にオーバーライドすることは不可能です。また、オーバーライドされることを目的としたequals() メソッド (抽象クラスのメソッドなど) は、具象クラスのequals() メソッドとは異なる内容で作成されます。このことを表すいくつかの例や、その詳細な理由については、『Effective Java Programming Language Guide』のItem 7 (参考文献を参照) をご覧ください。

改善の余地はありますか？

ハッシュをJavaクラス・ライブラリーのルートのオブジェクト・クラス内に組み込んだことは、きわめて賢明な設計上の妥協でした。それは、これにより、ハッシュに基づくコンテナーを非常に簡単に、かつより効率的に使用することができるからです。ただし、Javaクラス・ライブラリーにおけるハッシュと等価性についてのこの方針と実装方法に関しては、いくつか批判もありました。java.util のハッシュに基づくコンテナーは非常に便利で使いやすいものですが、きわめて高いパフォーマンスを要するアプリケーションには適していない場合があります。このようなコンテナーの多くは、まず変更されることはありませんが、ハッシュに基づくコンテナーの効率に大きく依存したプログラムを設計するにあたって、念頭に置いておくべきことがあります。それは、次のような批判です。

- ハッシュの範囲が非常に小さい:hashCode() の戻り値の型にlong でなくint を使用すると、ハッシュの衝突が発生する可能性が高くなります。
- ハッシュ値の分散度が低い: 短い文字列や短整数のハッシュ値は、それ自身が短整数であり、「近くの」他のオブジェクトのハッシュ値に近くなります。優れた機能を持つハッシュ関数であれば、ハッシュ値をハッシュ範囲内でより均等に分散させることができます。

- 定義されたハッシュ操作が存在しない:`String` や`List` などの一部のクラスでは、その複数の構成要素のハッシュ値を単一のハッシュ値に結びつけるために使用するハッシュ・アルゴリズムが定義されていますが、言語仕様としては、複数のオブジェクトのハッシュ値を1つの新しいハッシュ値に結び付けるための承認された方法が定義されていません。`List` や`String`、および、前述の独自のequals() およびhashCode() メソッドの作成で紹介したサンプル・クラス`A` で使用されている方法は、単純ですが、数学的な理想からは程遠いものです。また、クラス・ライブラリーでは、より高度な`hashCode()` の実装を簡単に作成できるような便利なハッシュ・アルゴリズムの実装もまったく提供されていません。
- 既に`equals()` をオーバーライドしているインスタンス化可能なクラスを継承する際の`equals()` の作成が難しい: 既に`equals()` をオーバーライドしているインスタンス化可能なクラスを継承するときに、「わかりやすい」方法で簡単に`equals()` を定義すると、`equals()` メソッドの対称性や推移性の要件を満たすことができません。これは、`equals()` をオーバーライドするときには、継承しようとしているクラスの構造や実装を詳細に理解する必要があるということであり、基底クラスのprivateフィールドをprotectedとして公開しなければならないために、優れたオブジェクト指向設計の原則に反することになる場合もあるということを意味します。

まとめ

`equals()` と`hashCode()` を矛盾なく定義することにより、ハッシュに基づくコレクションにおけるそのクラスのキーとしての有用性を高めることができます。等価性とハッシュ値の定義には、2つの方法があります。1つは`Object` でデフォルトとして提供されている、同一性に基づく方法であり、もう1つは、`equals()` と`hashCode()` の両方のオーバーライドが必要になる、状態に基づく方法です。オブジェクトの状態が変更されたときにそのハッシュ値も変更される可能性がある場合には、そのオブジェクトがハッシュ・キーとして使用されている間は状態の変更を許可しないようにする必要があります。

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。