Javaの理論と実践: ポイントは何処ですか?

浮動小数点数と10進数を扱うときの秘訣と落とし穴

Brian Goetz
Principal Consultant
Quiotix

2003年 1月 01日

特殊なタイミング・テストやベンチマーク・テストを除けば、プログラマーが固定小数点数や浮動小数点数を使用する機会はほとんどありません。Java言語とクラス・ライブラリーでは、IEEE 754浮動小数点 (`float` と`double`、およびラッパー・クラスの`Float` と`Double`)、および任意精度の10進数 (`java.math.BigDecimal`) という2つの非整数型がサポートされています。今月のJavaの理論と実践 では、Javaプログラムで非整数型を扱うときに直面する落とし穴についてBrian Goetz氏が説明します。

このシリーズの他の記事を見る

大抵のプロセッサーやプログラミング言語では浮動小数点演算をサポートしていますが、プログラマーがそれに注意を払う機会はほとんどありません。非整数型を使用することはほとんどないので、これは無理のないことです。科学的な計算や、時折見られるタイミング・テストやベンチマーク・テストを除けば、浮動小数点演算は見かけることさえありません。同様に、`java.math.BigDecimal` が提供する任意精度の10進数もアプリケーションで使用されることは少ないため、開発者に無視されているケースがほとんどです。しかし整数を中心としたプログラムに、突如、整数以外の表現が出てくることがあります。たとえば、JDBCではSQL`DECIMAL` 列の優先変換形式として`BigDecimal` が使用されます。

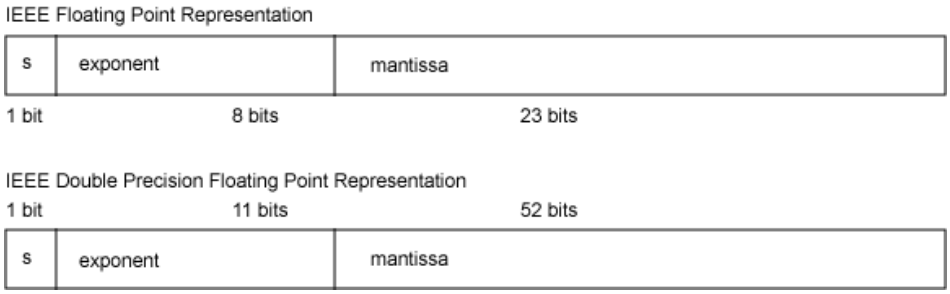
IEEE浮動小数点

Java言語では、2つの基本的な浮動小数点型`float` と`double`、およびそれらの型に対応するラッパー・クラスの`Float` と`Double` がサポートされています。これらの型は、32ビット浮動小数点と64ビット倍精度浮動小数点の2進小数に対するバイナリ規格を定義する、IEEE 754規格に基づいています。

IEEE 754では、浮動小数点数を基数2の指数表現を使った小数で表します。IEEEの浮動小数点数では、数値の符号に1ビット、指数部に8ビット、仮数 (または小数部) に23ビットが割り当てられています。指数部は符号付き整数として解釈されるので、正または負の指数を指定できます。小数

部は2進小数 (基数2) として表現されます。つまり、最高位ビットが$^1/_2$ (2^{-1})、2番目のビットが$^1/_4$ (2^{-2}) のように対応します。倍精度浮動小数点では、指数部に11ビット、仮数に52ビットが割り当てられています。図1は、IEEE浮動小数点のレイアウトを示しています。

図1. IEEE 754で定義されている浮動小数点レイアウト



指数表現で任意の数値を表す方法はいくつもあるため、浮動小数点数は、小数点の左側を1とする基数2の小数で表現するように正規化されています。この形式に合わせるため、必要に応じて指数を調整します。たとえば、1.25という数は、仮数部1.01と指数部0で表現されます。
$(-1)^0 * 1.01_2 * 2^0$

また10.0という数は、仮数部1.01と指数部3で表現されます。
$(-1)^0 * 1.01_2 * 2^3$

特殊な数値

この記法で許可されている標準的な値の範囲(`float` では1.4e-45から3.4028235e+38) に加えて、無限値 (Infinity)、負の無限値 (-Infinity)、`-0`、および非数 (NaN = not a number) を表す特殊な数値表現があります。これらの表現が準備されている理由は、算術オーバーフロー、負の数の平方根、`0` による除算などによって演算結果が正常な結果ではなくなった場合に、浮動小数点の値セットでその結果を表現するためです。

これらの特殊な数値には変わった特性があります。たとえば、`0` と`-0` は別個の値ですが、等しさを比較した場合、2つの値は等しいと解釈されます。ゼロ以外の数値を無限値で除算すると、結果は`0` になります。特殊値NaNには順序付け特性がないため、`==`、`<`、または`>` 演算子を使ってNaN値と他の浮動小数点値を比較した場合、結果は`false` になります。NaNを`f` とすると、`(f == f)` の結果も`false` になります。浮動小数点値とNaNを比較するときは、代わりに`Float.isNaN()` メソッドを使用します。表1は、無限値と非数のプロパティーを示しています。

表1. 特殊な浮動小数点数のプロパティー

式	結果
`Math.sqrt(-1.0)`	`-> NaN`
`0.0 / 0.0`	`-> NaN`
`1.0 / 0.0`	`-> Infinity`
`-1.0 / 0.0`	`-> -Infinity`
`NaN + 1.0`	`-> NaN`

`Infinity + 1.0`	`-> Infinity`
`Infinity + Infinity`	`-> Infinity`
`NaN > 1.0`	`-> false`
`NaN == 1.0`	`-> false`
`NaN < 1.0`	`-> false`
`NaN == NaN`	`-> false`
`0.0 == -0.0`	`-> true`

基本的なfloat型とラッパー・クラスのFloatでは比較動作が異なる

事態はさらにややこしくなります。NaNや`-0` を比較する規則が、基本的な`float` 型とラッパー・クラスの`Float` で違うのです。`float` 値の場合、2つのNaN値の等しさを比較すると結果は`false` になりますが、`Float.equals()` を使って2つのNaN`Float` オブジェクトを比較すると、結果は`true` になります。その理由は、結果を`true` にしないと、NaN`Float` オブジェクトを`HashMap` のキーとして使用できなくなるからです。同様に、float値では`0` と`-0` が等しいと解釈されますが、`Float.compareTo()` を使って`0` と`-0` を`Float` オブジェクトとして比較すると、`-0` は`0` より小さいと解釈されます。

浮動小数点に関するエラー

無限値、NaN、および`0` の特殊な特性があるため、一見正常に思える特定の変換や最適化も、浮動小数点数に適用すると実際に間違いを引き起こすことがあります。たとえば、`0.0-f` と`-f` は等しい値に見えますが、`f` が`0` だとすると等しくなくなります。表2は、同様の誤った解釈について示しています。

表2. 浮動小数点に関する誤った想定

式1	式2	以下の条件では式1と式2 は必ずしも等しくはない
`0.0 - f`	`-f`	fが`0`
`f < g`	`! (f >= g)`	fまたはgがNaN
`f == f`	`true`	fがNaN
`f + g - g`	`f`	gがinfinityまたはNaN

丸め誤差

浮動小数点演算では、厳密に正確な値になるのは稀です。`0.5` のような数値は2進小数 (基数2) を使って表現できますが(`0.5` は2^{-1} に等しい)、その他の`0.1` などの数値は正確に表現できません。そのため、浮動小数点演算の計算結果と正しい値がわずかにずれる、丸め誤差が発生することがあります。たとえば、以下の簡単な計算の結果は、`2.6` ではなく`2.600000000000001` となります。

```
double s=0;
for (int i=0; i<26; i++)
  s += 0.1;
System.out.println(s);
```

同様に、`.1*26` の結果と、`.1` を26回加算した結果は異なります。浮動小数点から整数にキャストする場合、丸め誤差は特に問題となります。これは、結果が整数値になるように「見える」演算でも、整数的な型へキャストすることによって整数以外の部分が破棄されてしまうためです。次の例をご覧ください。

```
double d = 29.0 * 0.01;
System.out.println(d);
System.out.println((int) (d * 100));
```

実際の結果は次のようになります。

```
0.29
28
```

この結果は、最初の予想と異なっていることでしょう。

浮動小数点数を比較するためのガイドライン

NaNの特殊な比較動作やほとんどの浮動小数点演算で起こる丸め誤差のため、浮動小数点値の比較演算の結果を解釈するときは注意が必要です。

最善の方法は、浮動小数点の比較演算を最初から回避することです。回避がいつでも可能とはかぎりませんが、浮動小数点比較演算の制限には常に注意しなければなりません。2つの浮動小数点数が等しいかどうかを判別する必要がある場合は、2つの値の差の絶対値を、事前に設定したイプシロン値と比較して、2つの値が "close enough" かどうかをテストする方が理にかなっています。(特に、基準として使用している測定スケールがわからない場合は、2つの値の差を単に比較するよりも "abs(a/b - 1) <epsilon" テストを使用した方が確実で効果的です。)1つの値をゼロと比較して、単に大きいか小さいかを判別することも危険です。ゼロよりわずかに大きいと「予想される」計算結果が、丸め誤差の計算による影響で、ゼロよりわずかに小さい値になることがあるためです。

NaNに順序付け特性がないことも、浮動小数点数を比較したときのエラーの要因になります。浮動小数点数を比較するときに無限値とNaNに関する落とし穴を避ける1つの方法は、無効な値を除外するのではなく、値の有効性を明示的にテストすることです。リスト1は、負以外の値だけを受け入れるプロパティーに対する、setterの2つの実装例を示しています。1番目の実装例はNaNを受け入れ、2番目の実装例ではNaNを受け入れません。2番目の実装例は、有効と思われる値の範囲を明示的にテストしているため適切な方法です。

リスト1. 負以外の浮動値を要求する2つの方法

```
// Trying to test by exclusion -- this doesn't catch NaN or infinity
public void setFoo(float foo) {
  if (foo < 0)
      throw new IllegalArgumentException(Float.toString(f));
    this.foo = foo;
}
// Testing by inclusion -- this does catch NaN
public void setFoo(float foo) {
  if (foo >= 0 && foo < Float.INFINITY)
    this.foo = foo;
  else
    throw new IllegalArgumentException(Float.toString(f));
}
```

正確な値を表すには浮動小数点を使用しない

ドルとセントを表す場合など、非整数値によっては正確さが求められる場合があります。浮動小数点数は厳密には正確な値ではありませんし、浮動小数点数を操作すると丸め誤差が生じます。そのため、通貨などの正確な量を表現するために浮動小数点を使用することはお勧めしません。セントの計算に浮動小数点を使用すると、結果は悲劇的なものになります。浮動小数点数が適しているのは、測定値のような値です。そもそも、このような値は根本的に不正確なのです。

小数値を扱うBigDecimal

JDK 1.3以降、Java開発者はもう1つの非整数値`BigDecimal` を使用できるようになりました。`BigDecimal` は、コンパイラーでの特別なサポートをされることのない標準クラスで、任意精度の10進数を表して計算が可能です。内部的には、`BigDecimal` は、任意精度の「スケールなしの整数値 (unscaled value)」と小数点以下の桁数を表す「スケール (scale)」係数で構成されます。このように、`BigDecimal` で表現される数値は`unscaledValue*10`$^{-scale}$ です。

`BigDecimal` 値の演算には、加算、減算、乗算、除算用のメソッドを使用します。`BigDecimal` オブジェクトは変更が不可能なため、これらのメソッドでは、それぞれ新しい`BigDecimal` オブジェクトを生成します。オブジェクト生成のオーバーヘッドが生じるため、`BigDecimal` は数値の計算が非常に多いものには向いていませんが、正確な10進数を表現するには適しています。通貨など、正確な量を表現するときには`BigDecimal` を使用してください。

すべてのequalsメソッドは等しく作成されていない

浮動小数点型と同様、`BigDecimal` にも落とし穴がいくつかあります。特に、数値の等しさを判別するために`equals()` メソッドを使用するときは注意が必要です。同じ数値を表していても、スケールが異なる2つの`BigDecimal` 値(たとえば`100.00` と`100.000`) は、`equals()` メソッドでは等しいと見なされません。ただし、`compareTo()` メソッドではこのような2つの値は等しいと見なされるので、2つの`BigDecimal` 値を数学的に比較するときは、`equals()` ではなく`compareTo()` を使用してください。

場合によっては、正確な結果を保持するのに任意精度の10進数演算では不十分なことがあります。たとえば`1` を`9` で除算すると、その答えは無限小数`.111111...` になります。このため、`BigDecimal` には、除算演算を実行したときの丸めを明示的に制御する機能があります。`movePointLeft()` メソッドでは、10の累乗による正確な除算がサポートされています。

変換型にBigDecimalを使用する

SQL-92には、固定小数点数を表現するための正確な数値型である`DECIMAL` データ型が含まれており、10進数に対して基本的な算術演算が実行されます。SQL言語によっては、この型の`NUMERIC` が優先される場合もあれば、小数点以下を2桁とする10進数として定義される`MONEY` データ型が含まれている場合もあります。

データベースの`DECIMAL` フィールドに数値を格納したり、`DECIMAL` フィールドから値を取得する場合、どのようにすれば正確な数値をやり取りすることができるでしょうか。浮動小数点と

小数の間で変換を行うと正確性が失われる可能性があるため、JDBCの`PreparedStatement` クラスや`ResultSet` クラスで提供されている`setFloat()` メソッドおよび`getFloat()` メソッドは使用できません。代わりに`PreparedStatement` および`ResultSet` の`setBigDecimal()` メソッドおよび`getBigDecimal()` メソッドを使用してください。

同様に、CastorのようなXMLデータ・バインディング・ツールは、10進数値の属性や要素 (XSDスキーマの基本データ型としてサポートされている) に対して、`BigDecimal` を使用してgetterおよびsetterを生成します。

BigDecimal数値の構築

`BigDecimal` 用のコンストラクターがいくつか提供されています。入力として倍精度浮動小数点を受け入れるもの、整数値とスケール係数を受け入れるもの、10進数の`String` 表現を受け入れるものなど、さまざまなコンストラクターが利用可能です。`BigDecimal(double)` コンストラクター (倍精度浮動小数点を受け入れる) を使用する場合は、プログラマーが気づかないうちに計算で丸め誤差が生じることがあるので注意が必要です。その場合は、代わりに整数を受け入れるコンストラクターを使用するか、`String` ベースのコンストラクターを使用してください。

`BigDecimal(double)` コンストラクターを誤って使用すると、JDBC`setBigDecimal()` メソッドに渡したときに、JDBCドライバーで予期せぬ例外が発生することがあります。例として、次のJDBC コード を見てみましょう。このコードでは、数値`0.01` を小数フィールドに格納します。

```
  PreparedStatement ps =
    connection.prepareStatement("INSERT INTO Foo SET name=?, value=?");
  ps.setString(1, "penny");
  ps.setBigDecimal(2, new BigDecimal(0.01));
  ps.executeUpdate();
```

`0.01` の倍精度近似値はスケールが大きな値になり、それが原因でJDBCドライバーまたはデータベースが戸惑う可能性があるため、JDBCドライバーによっては、無害に見えるこのコードを実行すると複雑な例外がスローされる場合があります。例外はJDBCドライバーで発生しますが、プログラマーが2進浮動小数点数の制限に注意しないかぎり、実際にコードのどの部分が間違っているかは判断できないでしょう。代わりに、`BigDecimal("0.01")` または`BigDecimal(1, 2)` のどちらかを使用して`BigDecimal` を構築すると、結果が正確な10進表現で表されるため、この問題を避けることができます。

まとめ

浮動小数点と10進数をJavaプログラムで使用することには、さまざまな落とし穴があります。浮動小数点と10進数は整数値ほど扱いやすくなく、結果が整数値または厳密に正確な値になると「予想される」浮動小数点計算で、実際にそのような結果が出されるとはかぎりません。浮動小数点数演算は、測定値など、根本的に不正確な値が関係した計算にとどめておくのが最善です。ドルやセントなど、固定小数点を表現する必要がある場合は、代わりに`BigDecimal` を使用してください。

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。