Java の新しい Math クラス: 第 1 回 実数

Elliotte Rusty Harold
Software Engineer
Cafe au Lait

2008年 10月 28日

この 2 回シリーズの記事では、Elliotte Rusty Harold が昔ながらの `java.lang.Math` クラスの「新しい」機能について調べます。第 1 回では純粋に数学的な関数に焦点を当て、第 2 回では浮動小数点数の演算用に設計された関数について調べます。

開発者が、あるクラスについて十分に理解してしまうと、そのクラスに注意を払わなくなる場合があります。もし皆さんが `java.lang.Foo` に関するドキュメントを作成したとして、Eclipse が関数を自動補完してくれるとしたら、皆さんは `java.lang.Foo` の Javadoc を読む必要があるでしょうか。私が java.lang.Math に関して経験したことが、まさにそれでした。それまで私は、自分が java.lang.Math を非常によく理解していると思いこんでいました。しかし最近、おそらく 5 年ぶりぐらいに java.lang.Math の Javadoc を読んだのですが、このクラスが約 2 倍のサイズになっており、私が聞いたこともないような 20 もの新しいメソッドが追加されていたのです。私がどれほど驚いたかを想像してみてください。どうやら、java.lang.Math をもう一度見直さなければならない時が来たようです。

Java™ 言語仕様のバージョン 5 では、`java.lang.Math` (そしてこれと同じメソッドを提供する、たちの悪い `java.lang.StrictMath`) に10 個の新しいメソッドが追加されており、Java 6 ではさらに 10 個のメソッドが追加されています。この記事では、提供されている関数のうち、より純粋に数学的な関数 (`log10` や `cosh` など) に焦点を当てます。第 2 回では、数学的な概念上のいわゆる実数の演算用に設計された関数ではなく、コンピューターが数値を扱う際の表現方法である浮動小数点数の演算用に設計された関数について調べます。

π や 0.2 といった数学的な概念上の実数と Java の `double` との間の違いは重要です。まず第 1 に、観念的に理想の数字は無限の正確さを持っていますが、Java での表現はある固定のビット数で表現できる範囲の正確さに制限されてしまいます。この違いは非常に大きい数字や非常に小さい数字を扱う際に重要です。例えば 2,000,000,001 (20 億 1) という数字は、`int` として正確に表現することはできますが、float として表現することはできません。`float` で表現した場合に最も近い数字は 2.0E9 (つまり 20 億) です。`double` の方がビット数は多いので少しはマシですが (このことから、ほとんど必ずと言ってよいほど `float` ではなく `double` を使う必要があります)、それでもやはり実際に得られる正確さには限界があります。

コンピューターによって (Java 言語や他の言語で) 演算する場合の第 2 の制限事項は、演算が 10 進数ではなく 2 進数をベースにしている点です。1/5 や 7/50 などの分数は 10 進数で正確に表現することができますが (それぞれ 0.2 と 0.14)、2 進数で表現した場合には循環小数になります。これは

1/3 を 10 進数で表現した場合に 0.3333333... になるのとまったく同じです。基数が 10 の場合、分母の素因数として 5 と 2 を持つ (そして他の素因数を持たない) 分数は、すべて正確に表現することができます。基数が 2 の場合には、分母が 2 の累乗になっている分数 (1/2、1/4、1/8、1/16 など) でないと正確に表現することはできません。

このような不正確さが出てきてしまうことが、そもそも math クラスが必要となる大きな理由の 1 つです。もちろん、三角関数やその他の関数を、+ や * などの標準的な演算子と単純なループのみを使ってテーラー級数展開によって定義することも可能です (リスト 1)。

リスト 1. テーラー級数を使って sin 関数を計算する

```
public class SineTaylor {

    public static void main(String[] args) {
        for (double angle = 0; angle <= 4*Math.PI; angle += Math.PI/8) {
            System.out.println(degrees(angle) + "\t" + taylorSeriesSine(angle)
                    + "\t" + Math.sin(angle));
        }
    }

    public static double degrees(double radians) {
        return 180 * radians/ Math.PI;
    }

    public static double taylorSeriesSine(double radians) {
        double sine = 0;
        int sign = 1;
        for (int i = 1; i < 40; i+=2) {
            sine += Math.pow(radians, i) * sign / factorial(i);
            sign *= -1;
        }
        return sine;
    }

    private static double factorial(int i) {
        double result = 1;
        for (int j = 2; j <= i; j++)  {
            result *= j;
        }
        return result;
    }
}
```

こうすると、最初はほんのわずかの差しか生じず、差がある場合も小数点の最後の桁が異なる程度です。

```
0.0       0.0       0.0
22.5       0.3826834323650897       0.3826834323650898
45.0       0.7071067811865475       0.7071067811865475
67.5       0.923879532511287        0.92387953251
90.0       1.0000000000000002       1.0
```

ところが、角度が増加するにつれてエラーが累積し始め、この単純な方法ではうまく行かなくなります。

```
630.0000000000003       -1.0000001371557132       -1.0
652.5000000000005       -0.9238801080153761       -0.9238795325112841
675.0000000000005       -0.7071090807463408       -0.7071067811865422
697.5000000000006       -0.3826922100671368       -0.3826834323650824
```

この場合のテーラー級数は実際のところ私の想像よりも正確でした。しかし角度が 360 度、720 度 (4π ラジアン)、あるいはそれ以上になると、テーラー級数で正確な計算をするためには累進的に項を増加する必要があります。`java.lang.Math` ではもっと高度なアルゴリズムが使用されており、この問題を回避しています。

またテーラー級数は、最近のデスクトップ・チップに組み込まれている sin 関数に比べると非効率です。sin やその他の関数を正確かつ高速に計算するためには、小さな誤差がいつのまにか大きな誤差になっていることのないように非常に注意深く設計されたアルゴリズムが必要です。これらのアルゴリズムは多くの場合、より高速に演算を処理するためにハードウェアの中に組みこまれています。例えば、この 10 年間に出荷されたほとんどすべての X86 チップには sin と cos がハードウェアで実装されているため X86 VM から単純に呼び出せばよく、それらよりもはるかに遅い、より基本的な演算をベースに計算する必要がなくなっています。HotSpot はこれらの命令を利用することによって三角関数の演算を劇的に高速化しています。

直角とユークリッド・ノルム

高校の幾何の授業では、誰もがピタゴラスの定理を学びます。つまり直角三角形の斜辺の長さの 2 乗は他の 2 辺の 2 乗の和に等しく、$c^2 = a^2 + b^2$ です。

辛抱強く大学に進み、物理や高度な数学を学んだ人達は、この等式が直角三角形を表す以外にも何度も登場することを学びました。例えばこの公式はユークリッド・ノルムの 2 乗の R^2 としても登場し、2 次元ベクトルの長さでもあり、三角不等式の一部でもあり、その他にも多くの例があります。(実際には、これらはすべて、同じものに対して異なる見方をしているにすぎません。つまり、ユークリッドの定理は最初に受ける印象以上にずっと重要なのです)。

Java 5 では、まさにこの計算を行うための `Math.hypot` 関数が追加されており、なぜライブラリーが便利なのかを示す好例となっています。この関数の単純な使い方は、次のとおりです。

```
public static double hypot(double x, double y){
  return Math.sqrt (x*x + y*y);
}
```

実際のコードは、もう少し複雑です (リスト 2)。リスト 2 を見て最初に気付くであろうことは、このコードが最大のパフォーマンスを得るためにネイティブの C で作成されている点です。そしておそらく次に、この計算の中で起こりうる誤差を最小限にとどめるためにかなり苦労していることに気づくはずです。実際、`x` と `y` の相対的な大きさの違いによって異なるアルゴリズムが選択されています。

リスト 2. `Math.hypot` を実装した実際のコード

```
/*
 * ====================================================
 * Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
 *
 * Developed at SunSoft, a Sun Microsystems, Inc. business.
 * Permission to use, copy, modify, and distribute this
 * software is freely granted, provided that this notice
 * is preserved.
 * ====================================================
 */
```

```
#include "fdlibm.h"

#ifdef __STDC__
	double __ieee754_hypot(double x, double y)
#else
	double __ieee754_hypot(x,y)
	double x, y;
#endif
{
	double a=x,b=y,t1,t2,y1,y2,w;
	int j,k,ha,hb;

	ha = __HI(x)&0x7fffffff;	/* high word of  x */
	hb = __HI(y)&0x7fffffff;	/* high word of  y */
	if(hb > ha) {a=y;b=x;j=ha; ha=hb;hb=j;} else {a=x;b=y;}
	__HI(a) = ha;	/* a <- |a| */
	__HI(b) = hb;	/* b <- |b| */
	if((ha-hb)>0x3c00000) {return a+b;} /* x/y > 2**60 */
	k=0;
	if(ha > 0x5f300000) {	/* a>2**500 */
	   if(ha >= 0x7ff00000) {	/* Inf or NaN */
	       w = a+b;			/* for sNaN */
	       if(((ha&0xfffff)|__LO(a))==0) w = a;
	       if(((hb^0x7ff00000)|__LO(b))==0) w = b;
	       return w;
	   }
	   /* scale a and b by 2**-600 */
	   ha -= 0x25800000; hb -= 0x25800000;	k += 600;
	   __HI(a) = ha;
	   __HI(b) = hb;
	}
	if(hb < 0x20b00000) {	/* b < 2**-500 */
	    if(hb <= 0x000fffff) {	/* subnormal b or 0 */
	       if((hb|(__LO(b)))==0) return a;
	       t1=0;
	       __HI(t1) = 0x7fd00000;	/* t1=2^1022 */
	       b *= t1;
	       a *= t1;
	       k -= 1022;
	    } else {		/* scale a and b by 2^600 */
	        ha += 0x25800000;	/* a *= 2^600 */
	       hb += 0x25800000;	/* b *= 2^600 */
	       k -= 600;
	          __HI(a) = ha;
	          __HI(b) = hb;
	    }
	}
    /* medium size a and b */
	   w = a-b;
	   if (w>b) {
	       t1 = 0;
	       __HI(t1) = ha;
	       t2 = a-t1;
	       w  = sqrt(t1*t1-(b*(-b)-t2*(a+t1)));
	   } else {
	       a  = a+a;
	       y1 = 0;
	       __HI(y1) = hb;
	       y2 = b - y1;
	       t1 = 0;
	       __HI(t1) = ha+0x00100000;
	       t2 = a - t1;
	       w  = sqrt(t1*y1-(w*(-w)-(t1*y2+t2*b)));
	   }
	   if(k!=0) {
	       t1 = 1.0;
	       __HI(t1) += (k<<20);
```

```
            return t1*w;
        } else return w;
}
```

実際には、この特定の関数を使うのか、あるいはこの関数に似た他のいくつかの関数のうちの 1 つを使うのかは、プラットフォーム上の JVM の詳細に依存します。しかし Sun の標準 JDK で呼び出されるのは、おそらくこのコードです。(この JDK の他の実装では、必要に応じてこのコードを自由に改善することができます。)

このコード (そして、Sun の Java Development Library の中にある他のネイティブ math コードの大部分) は、15 年ほど前に Sun で作成されたオープンソースの `fdlibm` ライブラリーから引用したものです。このライブラリーは、たとえ少しばかりパフォーマンスを犠牲にしても IEEE754 浮動小数点演算を正確に実装するように、そして非常に正確な計算を行うように設計されています。

10 を底とする対数

対数は、与えられた値を生成するために底数を何乗する必要があるかを表します。つまり対数は `Math.pow()` 関数の逆です。10 を底とする対数はエンジニアリング·アプリケーションでよく登場します。e を底とする対数 (自然対数) は複利計算や数々の科学アプリケーションや数学アプリケーションに登場します。2 を底とする対数はアルゴリズムの分析によく登場します。

`Math` クラスには Java 1.0 の時から自然対数関数があります。つまり x という引数を与えると、自然対数は x という値を生成するために e を何乗する必要があるかという乗数を返します。残念なことに、Java 言語では (そして C や Fortran や Basic なども) 自然対数関数には log() という誤った名前が付けられています。私が今までに読んだことのあるどの数学の教科書でも、`log` は 10 を底とする対数であり、ln は e を底とする対数であり、lg は 2 を底とする対数になっています。この問題を今から修正することはできませんが、Java 5 では対数の底を e ではなく 10 とする `log10()` 関数を追加しています。

リスト 3 は 1 から 100 までの整数に対して 2、10、e を底とする対数を出力する簡単なプログラムです。

リスト 3. 1 から 100 までの整数に対してさまざまな数を底とする対数

```
public class Logarithms {
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            System.out.println(i + "\t" +
                               Math.log10(i) + "\t" +
                               Math.log(i) + "\t"  +
                               lg(i));
        }
    }

    public static double lg(double x) {
        return Math.log(x)/Math.log(2.0);
    }
}
```

この出力の最初の 10 行を以下に示します。

```
1    0.0                                0.0                                0.0
2    0.3010299956639812  0.6931471805599453   1.0
3    0.47712125471966244 1.0986122886681096  1.584962500721156
4    0.6020599913279624  1.3862943611198906   2.0
5    0.6989700043360189  1.6094379124341003   2.321928094887362
6    0.7781512503836436  1.791759469228055     2.584962500721156
7    0.8450980400142568  1.9459101490553132   2.807354922057604
8    0.9030899869919435  2.0794415416798357   3.0
9    0.9542425094393249  2.1972245773362196   3.1699250014423126
10  1.0                              2.302585092994046     3.3219280948873626
```

`Math.log10()` には、対数関数に関するおなじみの注意点があります。つまり 0 または負の数の対数を取ろうとすると NaN が返されます。

立方根

私はこれまでの人生で立方根を取る必要に迫られたことがなく、しかも私は代数や幾何を日常的に使用しており、時によると微積分や微分方程式、さらには抽象代数まで使用するという稀な人間の 1 人です。そのため、次に説明するこの関数の有用性は私には思い当たりません。とはいえ、立方根を取らなければならないという予期せぬ事態にどこかで陥った場合には、(Java 5 の時点で) `Math.cbrt()` メソッドを使うことができます。リスト 4 は -5 から 5 までの整数の立方根を取る方法を示しています。

リスト 4. -5 から 5 までの立方根

```
public class CubeRoots {
    public static void main(String[] args) {
        for (int i = -5; i <= 5; i++) {
            System.out.println(Math.cbrt(i));
        }
    }
}
```

この出力は次のとおりです。

```
-1.709975946676697
-1.5874010519681996
-1.4422495703074083
-1.2599210498948732
-1.0
0.0
1.0
1.2599210498948732
1.4422495703074083
1.5874010519681996
1.709975946676697
```

この出力を見ると、立方根には平方根に比べて 1 つ便利な特徴があることがわかります。それは、すべての実数には必ず 1 つだけ実数の立方根があることです。この関数が NaN を返すのは引数が NaN の場合のみです。

双曲線三角関数

双曲線三角関数と双曲線との関係は、三角関数と円との関係と同じです。つまりデカルト平面上でとりうる任意の t という値に対して次のような点をプロットすることを考えてみてください。

```
x = r cos(t)
y = r sin(t)
```

すると、半径 r の円が描かれます。一方、sin と cos ではなく次のように sinh と cosh を使うとしたらどうでしょう。

```
x = r cosh(t)
y = r sinh(t)
```

すると、原点との最短距離が r である直角双曲線が描かれます。

別の考え方をすると、sin(x) は $(e^{ix} - e^{-ix})/2i$ と書くことができ、cos(x) は $(e^{ix} + e^{-ix})/2$ と書くことができますが、sinh と cosh の場合はこれらの公式から虚数の単位を除いたものです。つまり sinh(x) = $(e^{x} - e^{-x})/2$ であり、cosh(x) = $(e^{x} + e^{-x})/2$ です。

Java 5 では、`Math.cosh()`、`Math.sinh()`、`Math.tanh()` という 3 つがすべて追加されています。双曲線逆三角関数 (acosh、asinh、atanh) はまだ含められていません。

自然界においては、cosh(z) は両端を固定して垂らしたロープの形 (カテナリーとして知られています) を表す等式です。リスト 5 は `Math.cosh` 関数を使ってカテナリーを描画する簡単なプログラムです。

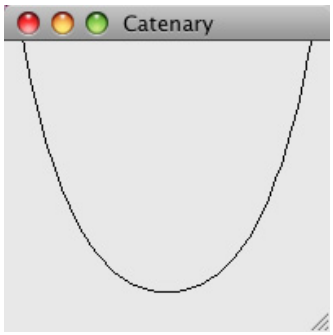
リスト 5. `Math.cosh` 関数を使ってカテナリーを描画する

```
import java.awt.*;

public class Catenary extends Frame {

    private static final int WIDTH = 200;
    private static final int HEIGHT = 200;
    private static final double MIN_X = -3.0;
    private static final double MAX_X = 3.0;
    private static final double MAX_Y = 8.0;

    private Polygon catenary = new Polygon();

    public Catenary(String title) {
        super(title);
        setSize(WIDTH, HEIGHT);
        for (double x = MIN_X; x <= MAX_X; x += 0.1) {
            double y = Math.cosh(x);
            int scaledX = (int) (x * WIDTH/(MAX_X - MIN_X) + WIDTH/2.0);
            int scaledY = (int) (y * HEIGHT/MAX_Y);
            // in computer graphics, y extends down rather than up as in
            // Caretesian coordinates' so we have to flip
            scaledY = HEIGHT - scaledY;
            catenary.addPoint(scaledX, scaledY);
        }
    }

    public static void main(String[] args) {
        Frame f = new Catenary("Catenary");
        f.setVisible(true);
    }

    public void paint(Graphics g) {
        g.drawPolygon(catenary);
    }
```

```
}
```

図 1 は描画された曲線を示しています。

図 1. デカルト平面上のカテナリー曲線



また sinh 関数、cosh 関数、tanh 関数はどれも、特殊相対性理論や一般相対性理論のさまざまな計算にも登場します。

符号属性

`Math.signum` 関数は、正の数を 1.0 に、負の数を -1.0 に、ゼロをゼロに変換します。要するに、この関数は数字から符号のみを抽出します。これは `Comparable` インターフェースを実装する際に便利です。

この関数には、型を維持するために `float` 型や `double` 型も用意されています。こうした自明の関数がある理由は、浮動小数点演算や NaN、正のゼロ、負のゼロなどの特別なケースを処理するためです。NaN もゼロのように扱われ、また正のゼロと負のゼロは、正のゼロと負のゼロを返す必要があります。例えば、この関数を単純に実装すると次のようになります (リスト 6)。

リスト 6. バグのある `Math.signum` の実装

```
public static double signum(double x) {
  if (x == 0.0) return 0;
  else if (x < 0.0) return -1.0;
  else return 1.0;
}
```

しかしこれではバグがあるので、このメソッドではまず、負のゼロをすべて正のゼロに変換する必要があります。(確かに負のゼロと言うと少し奇妙ですが、IEEE 754 の仕様では必要なのです。) そして次に、NaN が正であると宣言します。実際の実装では、こうした奇妙なコーナー・ケース (稀な場合) をより洗練された慎重な方法で処理しています (リスト 7)。

リスト 7. 実際の、適切な `Math.signum` の実装

```
public static double signum(double d) {
    return (d == 0.0 || isNaN(d))?d:copySign(1.0, d);
}

public static double copySign(double magnitude, double sign) {
    return rawCopySign(magnitude, (isNaN(sign)?1.0d:sign));
}

public static double rawCopySign(double magnitude, double sign) {
    return Double.longBitsToDouble((Double.doubleToRawLongBits(sign) &
                                    (DoubleConsts.SIGN_BIT_MASK)) |
                                    (Double.doubleToRawLongBits(magnitude) &
                                    (DoubleConsts.EXP_BIT_MASK |
                                     DoubleConsts.SIGNIF_BIT_MASK)));
}
```

少ない努力で多くを得る

最も効率的なコードは、皆さんが決して作成することもないようなコードです。既にエキスパートが作成したものを自分で作成する必要はありません。`java.lang.Math` 関数を使うコードは、古いコードも新しいコードも、皆さん自身が作成するどんなコードよりも高速で効率的であり、しかも正確なのです。積極的に java.lang.Math 関数を使ってください。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold はニューオーリンズ出身であり、時たま、おいしいガンボ (gumbo: オクラ入りのスープ) を食べに帰っています。ただし現在はアーヴィン近郊の University Town Center に、妻の Beth と猫の Charm (charmed quarkからとりました) と Marjorie（ 義理の母の名前からとりました ）と一緒に住んでいます。彼の Webサイト Cafe au Lait は、インターネット上で最も人気のある独立系 Java サイトの 1 つです。また、そこから派生した Cafe con Leche は、最も人気のある XML サイトの一つです。彼の最近の著作には『Refactoring HTML』があります。