java.lang.Math简单介绍

# Math类与StrictMap类

Math类与Strict类都存在于java.lang包中，作用都是提供一些常用的数字操作函数。

Math类里面许多数学函数方法是直接调用的StrictMath类里面的方法，而这些函数方法在StrictMath里面的形式是调用**本地的非Java代码的接口**，使用**native**关键字进行修饰，利用C语言实现的，使用C代码来实现这些基本的函数好处就是**提高运行的效率**。

**By default many of the Math methods simply call the equivalent method in StrictMath for their implementation**.

这句话说明了Math类中的函数默认都是调用了StrictMath中对应的方法。

如：Math中的源码：



而StrictMath中的方法一般都是native方法。



两个类的**相同点**：都是实现了基本的数学方法，都有互相调用的地方

**不同点**：两个类的所有相同方法中，Math类调用了StrictMath类的方法，而且都是用native修饰，C代码实现的方法。

性能比较：Math的效率高，而StrictMath可以保障在不同的平台上运行都得到相同的结果。

在Math类里面的函数利用**计算机浮点单元的例程**来达到**最快的性能**。如果得到**完全的可预测的结果**，比最快的性能更重要的话，应该使用**StrictMath**方法。它从**“自由分布数学库”（fdlibm）**实现其运算法则，这样保证了在所有平台上都可以得到相同的结果。

# java.lang.Math类

public final class **Math** extends **Object**

**很简单的一个工具类。**

**功能**：The class Math contains methods for **performing basic numeric operations** such as the elementary exponential, logarithm, square root, and trigonometric functions.

Math类提供了一些关于数字基本操作的静态方法，如指数、对数、平方根、三角函数等。

Unlike some of the numeric methods of class **StrictMath**, all implementations of the equivalent functions of class Math are not defined **to return the bit-for-bit same results**. This relaxation permits **better-performing implementations** where strict reproducibility is not required.

By default many of the Math methods simply call the equivalent method in **StrictMath** for their implementation. **Code generators** are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of Math methods. Such higher-performance implementations still must conform to the specification for Math.

The quality of implementation specifications concern two properties, **accuracy of the returned result** and **monotonicity(单调性，单一性) of the method**. Accuracy of the floating-point Math methods is measured in terms of **ulps**, units in the last place. For a given floating-point format, an ulp of a specific real number value is the distance between the two floating-point values bracketing that numerical value. When discussing the accuracy of a method as a whole rather than at a specific argument, the number of ulps cited is for the worst-case error at any argument. If a method always has an error less than 0.5 ulps, the method always returns the floating-point number nearest the exact result; such a method is correctly rounded. A correctly rounded method is generally the best a floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the Math class, a larger error bound of 1 or 2 ulps is allowed for certain methods. **Informally**, with a 1 ulp error bound, when the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 ulp errors are required to be semi-monotonic: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

The platform uses signed two's complement integer arithmetic with int and long primitive types. The developer should choose the primitive type to ensure that arithmetic operations consistently produce correct results, which in some cases means the operations will not overflow the range of values of the computation. The best practice is to choose **the primitive type** and **algorithm to avoid overflow**. In cases where the size is int or long and overflow errors need to be detected, the methods **addExact**, **subtractExact**, **multiplyExact**, and **toIntExact** throw an ArithmeticException when the results overflow. For other arithmetic operations such as divide, absolute value, increment, decrement, and negation overflow occurs only with a specific minimum or maximum value and should be checked against the minimum or maximum as appropriate.

## 两个静态属性

### 自然常数e

static **double** **E**

The double value that is closer than any other to **e**, the base **of the natural logarithms**.

圆周率π

### static **double** PI

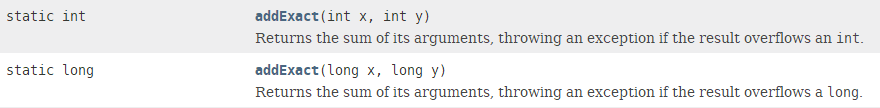
The double value that is closer than any other to **pi**, the ratio of the circumference of a circle to its diameter.

## 提供的数学函数有

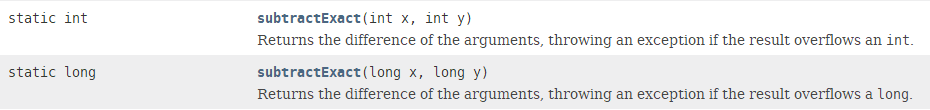
abs、

## 加减乘除

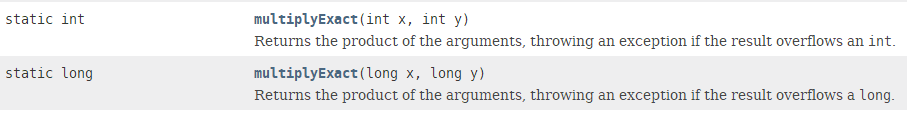
### 加法



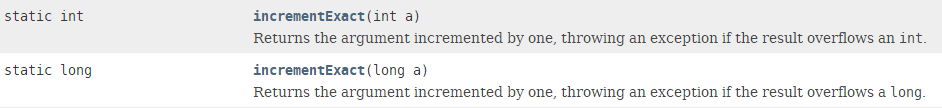
### 减法

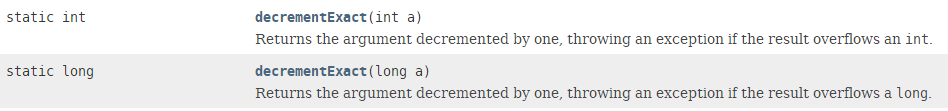


### 乘法multiplyExact



### incrementExact:增加1；incrementExact：减1

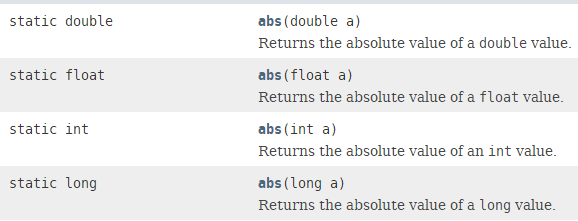




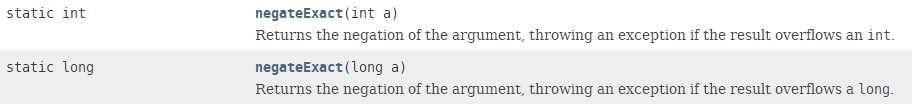
### hypot



### abs取绝对值

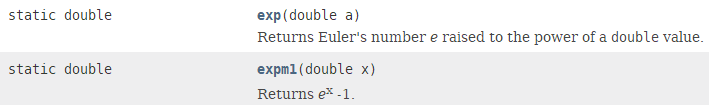


### 相反数

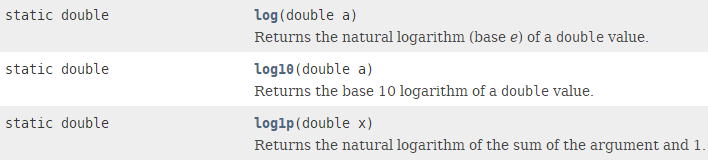


## 对数指数运算

### 指数函数exp



### 对数函数



## 幂与方根



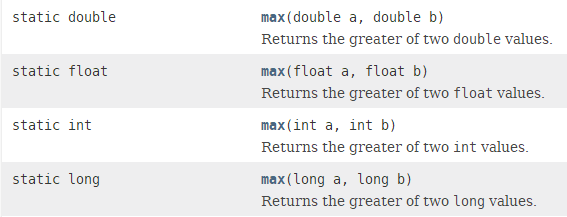
pow：幂



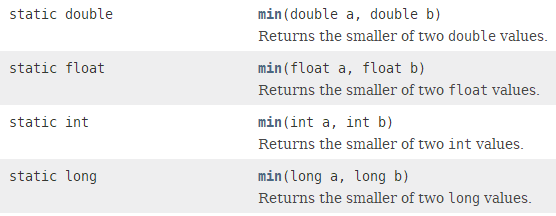


## 最大值最小值

### 两者中的最大值max

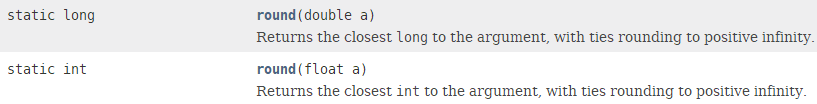


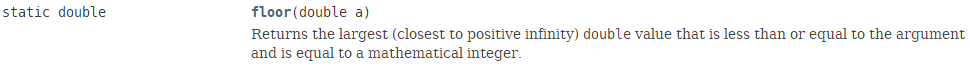
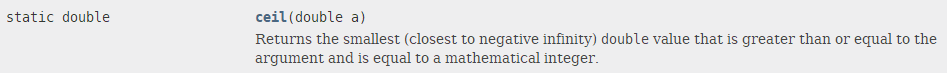
### 两者中的最小值min



## 取舍函数

floor、ceil、round

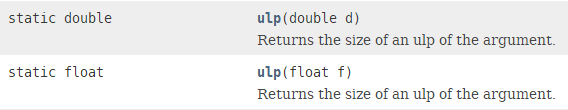


## ulp函数

### ulp函数:返回参数对应的ulp。

ulp的意义：**假设在float 2.0和3.0之间有8,388,609个数，那么在2.0和3.0之间的数的ulp就是1.0/8,388,609约等于0.0000001。**

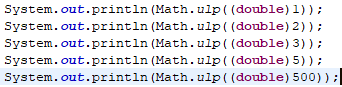
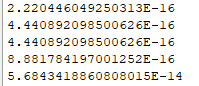




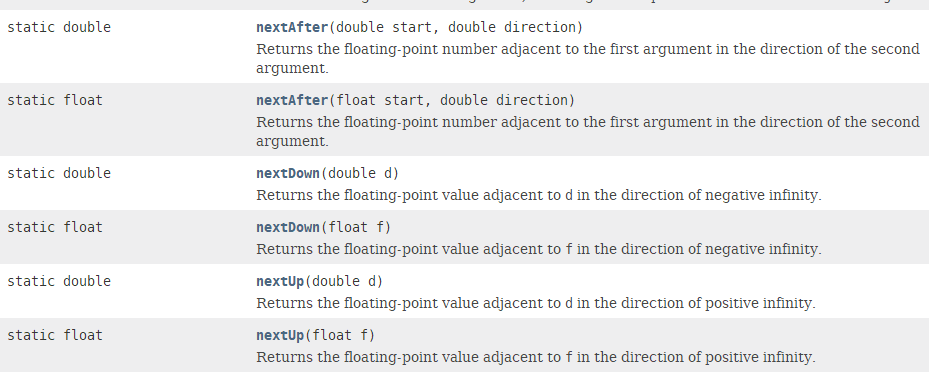
1.1920929E-7

2.220446049250313E-16

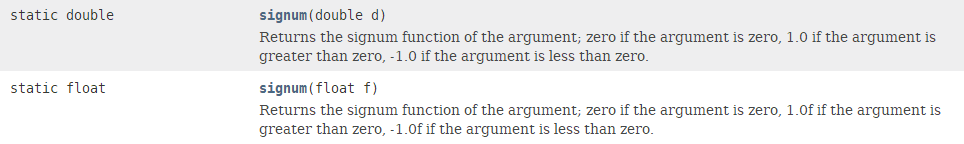
有些数对应的ulp相同，有些不同：

### next



## 符号函数



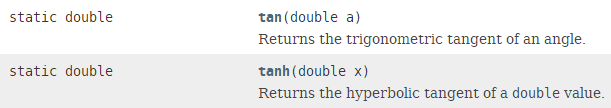
## 三角函数：

### toRadians：角度转弧度

static double **toRadians**(double angdeg)

Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

tan、tanh、sin、sinh、cos、cosh



# Math的ulp函数理解

如果要理解什么是ulp，先要了解在计算机中保存的数和我们在数学上认为的数是不一样的；

比方说2.0和3.0之间有多少个数，在数学中是无限的，但是在计算机中是有限的，因为计算机需要用一堆字节来表示double或者float，但是因为计算机表示不了无限的数（因为没有无限内存）。

**所以就有了ulp，假设在float 2.0和3.0之间有8,388,609个数，那么在2.0和3.0之间的数的ulp就是1.0/8,388,609约等于0.0000001。**

你如果想知道某一个具体的double或float的先一个或者上一个数字是什么可以使用函数

public static double **nextAfter**(float start, float direction)

public static double **nextAfter**(double start, double direction)

# StrictMath

public final class **StrictMath** extends Object

属性和方法名称与Math几乎完全相同。

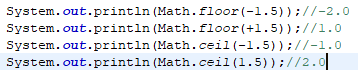
StrictMath是从JDK1.3推出的；Math是JDK1.0就存在。

但是，Math类的方法大部分都是调用StrictMath的。

# round函数

对于ceil、floor函数都好判断其值，返回**int**或**double**类型。

ceil向上取整，floor向下取整。



但是，对于round函数：返回**int**或**long**类型。

**可以理解为大于0的数，四舍五入；小于0的数，五舍六入。**



**特殊的：round函数**

**StrictMath**中的round函数是通过调用Math的round函数实现的。



Math中的round函数是java代码实现的，源代码：

