Числа с плавающей запятой и их особенности

Многие программисты годами пишут свои программы, не понимая, что такое числа с плавающей запятой, и чем они отличаются от "обычных", целых чисел. Это не мешает им создавать хорошие программы. Но в конце концов каждый сталкивается с "необъяснимым" явлением:

```php
$a = 1.1 - 1;
$b = 0.1;
if ($a == $b) {
  echo "$a равно $b";
} else {
  echo "$a не равно $b";
}
```

Эта программа печатает "0.1 не равно 0.1". В чём дело? Напрашивается вывод, что в языке программирования что-то не в порядке. В сети можно найти немало переписок с разработчиками языков о подобных "ошибках". На самом же деле, этот пример демонстрирует некоторые важные свойства чисел с плавающей запятой.

Происхождение названия

Название **«плавающая запятая»** происходит от того, что запятая в позиционном представлении числа (десятичная запятая, или, для компьютеров, двоичная запятая — далее по тексту просто запятая) может быть помещена где угодно относительно цифр в строке. Это положение запятой указывается отдельно во внутреннем представлении. Таким образом, представление числа в форме с плавающей запятой может рассматриваться как компьютерная реализация экспоненциальной записи чисел.

Преимущество использования представления чисел в формате с плавающей запятой над представлением в формате с фиксированной запятой (и целыми числами) состоит в том, что можно использовать существенно больший диапазон значений при неизменной относительной точности.

Например, в форме **с фиксированной запятой** число, занимающее 6 разрядов в целой части и 2 разряда после запятой, может быть представлено в виде 123 456,78. В свою очередь, в формате с плавающей запятой в тех же 8 разрядах можно записать числа 1,2345678; 1 234 567,8; 0,000012345678; 12 345 678 000 000 000 и так далее, но для этого необходимо иметь дополнительное двухразрядное поле для записи показателей степени 10 от 0 до 16, при этом общее число разрядов составит 8+2=10.

Скорость выполнения компьютером операций с числами, представленными в форме с плавающей запятой, измеряется во FLOPS (от англ. *floating-point operations per second* — «[количество] операций с плавающей запятой в секунду»), и является одной из основных единиц измерения быстродействия вычислительных систем.

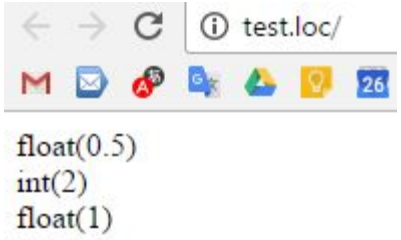
Как узнать, что используются числа с плавающей запятой?

В языках программирования со строгой типизацией существуют, как правило, специальные типы данных для чисел с плавающей запятой (float/double/long double в Си, single/double/extended в Паскале). Если в вычислении участвует хотя бы одна переменная или константа с плавающей запятой, все другие числа тоже преобразовываются к этому типу.

В языках без строгой типизации, как Perl, PHP или JavaScript, заметить использование чисел с плавающей запятой сложнее. Для программиста все числа выглядят одинаково, переключение с целочисленных типов на типы с плавающей запятой происходит автоматически.

Можно исходить из того, что используются операции для чисел с плавающей запятой, если какая-нибудь из участвующих переменных содержит дробную часть или её значение выходит за пределы диапазона целых чисел. Но бывают и случаи, когда числа с плавающей запятой используются для целочисленных значений:

```
2   $a = 0.5;
3   var_dump($a);
4   $b = 2;
5   var_dump($b);
6   $b = $a * $b;
7   var_dump($b);
```



Здесь переменная $a равна единице (это покажет и сравнение, в отличие от примера в начале), но её значение всё равно хранится как число с плавающей запятой потому, что её значение *раньше* содержало дробную часть.

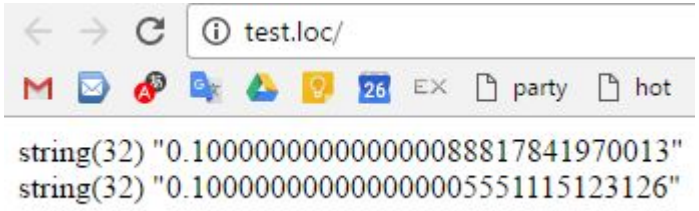
Откуда берётся неточность?

Основная причина неточности при использовании чисел с плавающей запятой в том, что компьютер не может работать с бесконечными дробями, которые мы знаем из математики — для них понадобилось бы бесконечное количество памяти. Это и понятно, мы тоже округляем числа до какого-то знака, когда имеем дело с десятичными дробями. Но это не объясняет приведённого в начале статьи примера — ведь там всего один знак после запятой?

Существует ещё один фактор — компьютер считает не в десятичной системе, а в двоичной. А если представить 0.1 как двоичную дробь, то она окажется периодической: 0.0(0011). Соответственно, в памяти компьютера число 0.1 представлено как 1.10011001100110011001100110011001100110011001100110011010b * 2^(-4). Обратите внимание на округление в конце числа. Если перевести его обратно в десятичную систему, то получится 0.1000000000000000555111512.

Почему тогда показывается не это число, а 0.1? Дело в том, что числа с плавающей запятой на выводе всегда округляются. Если использовать функцию php number_format, то можно вывести до необходимое количество значащих знаков, и тогда мы вдруг увидим 0.1000000000000005. В некоторых браузерах метод Number.toPrecision() JavaScript'a позволяет выводить числа даже с пятьюдесятью значащими знаками.

```php
$a = 1.1 - 1;
$b = 0.1;
var_dump(number_format ($a,30));
var_dump(number_format($b,30));
```



Так почему всё-таки 1.1 − 1 не равно 0.1? Если посмотреть значение числа 1.1 − 1, то мы увидим 0.100000000000000088817841970013. Оно, очевидно, не равно компьютерному представлению числа 0.1, хотя при стандартном округлении и выглядит точно так же. Разница объясняется тем, что мы считали с округлённой версией числа 1.1.

---

* number_format — Форматирует число с разделением групп (http://php.net/manual/ru/function.number-format.php)

Как бороться с погрешностями?

Если использовать числа с плавающей запятой, то погрешность результатов оценить сложно. До сих пор не существует удовлетворительной математической теории, которая позволяла бы это делать.

* Как утверждает официальный сайт PHP - Никогда не доверяйте точности чисел с плавающей точкой до последней цифры, и не проверяйте напрямую их равенство. **Если вам действительно необходима высокая точность, используйте математические функции PHP произвольной точности** (http://php.net/manual/ru/ref.bc.php) и **gmp-функции** (http://php.net/manual/ru/ref.gmp.php).

---

http://php.net/manual/ru/language.types.float.php

BC Math Функции

bcadd — Сложить 2 числа произвольной точности
bccomp — Сравнение двух чисел произвольной точности
bcdiv — Операция деления для чисел произвольной точности
bcmod — Получает остаток от деления чисел с произвольной точностью
bcmul — Умножение двух чисел с произвольной точностью
bcpow — Возведение в степень чисел с произвольной точностью
bcpowmod — Возводит одно число в степень другого и возвращает остаток от деления результата на третье число
bcscale — Задает количество чисел после десятичной точки по умолчанию для всех bc math функций.
bcsqrt — Извлекает квадратный корень из числа с заданной точностью
bcsub — Вычитает одно число из другого с заданной точностью

```
2   var_dump(number_format(1.1 - 1, 30));
3   var_dump(bcsub(1.1, 1, 30));
```

```
:string '0.100000000000000088817841970013' (length=32)
:string '0.100000000000000000000000000000' (length=32)
```

В JavaScript тоже есть выход

Округление

Одна из самых частых операций с числом – округление. В JavaScript существуют целых 3 функции для этого.

```
6   alert( Math.floor(3.1) );  // 3
7   alert( Math.ceil(3.1) );   // 4
8   alert( Math.round(3.1) );  // 3
```

Math.floor - Округляет вниз
Math.ceil - Округляет вверх
Math.round - Округляет до ближайшего целого

Округление до заданной точности

Для округления до нужной цифры после запятой можно умножить и поделить на 10 с нужным количеством нулей. Например, округлим 3.456 до 2-го знака после запятой:

```
6   var n = 3.456;
7   alert( Math.round(n * 100) / 100 ); // 3.456 -> 345.6 -> 346 -> 3.46
```

Таким образом можно округлять число и вверх и вниз.

num.toFixed(precision)

Существует также специальный метод num.toFixed(precision), который округляет число num до точности precision и возвращает результат в виде строки:

```
6   var n = 12.34;
7   alert( n.toFixed(1) ); // "12.3"
8   alert( n.toFixed(5) ); // "12.34000", добавлены нули до 5 знаков после запятой
```

Метод toFixed не эквивалентен Math.round!

Например, произведём округление до одного знака после запятой с использованием двух способов: toFixed и Math.round с умножением и делением:

```
9   var price = 6.35;
10  alert( price.toFixed(1) ); // 6.3
11  alert( Math.round(price * 10) / 10 ); // 6.4
```

Как видно, результат разный! Вариант округления через Math.round получился более корректным, так как по общепринятым правилам 5 округляется вверх. А toFixed может округлить его как вверх, так и вниз. Почему? По тем же выше описанным причинам погрешности вычислений.

```
17   alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

Есть два способа сложить 0.1 и 0.2:

Сделать их целыми, сложить, а потом поделить:

Это работает, т.к. числа 0.1*10 = 1 и 0.2*10 = 2 могут быть точно представлены в двоичной системе.

```
6   alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
```

Сложить, а затем округлить до разумного знака после запятой. Округления до 10-го знака обычно бывает достаточно, чтобы отсечь ошибку вычислений:

```
13   var result = 0.1 + 0.2;
14   alert( +result.toFixed(10) ); // 0.3
```

Бесконечность и прочие вкусности

Для чисел с плавающей запятой определены несколько специальных значений, которые весьма непривычны для программистов, привыкших к целочисленным операциям. **Так, если взять самое большое целое число и прибавить к нему единицу, произойдёт переполнение, и число станет отрицательным. Если же прибавить единицу к самому большому числу с плавающей запятой, то не произойдёт ровным счётом ничего;** в результате мы получим то же самое число. Переполнения можно добиться, к примеру, умножив это число на два. Но результат будет несколько необычным — "число" `Inf` (от англ. *infinity* = бесконечность). Аналогичным образом можно получить отрицательную бесконечность — `-Inf`.

Бесконечность получается и при делении на ноль, причём и здесь она может быть как положительной, так и отрицательной (никакого исключения, как при работе с целыми числами, не возникает). И с ней действительно можно решать! Так, если разделить любое число на бесконечность, получится ноль. Произведение двух бесконечностей опять даёт бесконечность, как и сумма бесконечностей с одинаковым знаком.

А вот сумма бесконечностей с разными знаками не определена, результатом получается `NaN`, другое специальное значение (от англ. *Not a Number* = не число). То же самое выйдет, если попытаться умножить бесконечность на ноль или поделить ноль на ноль. В некоторых языках программирования `NaN` является ещё и результатом неудачного преобразования строки в число. С `NaN` тоже можно решать, но результат любой операции будет опять же `NaN`.

Ну и ещё одно необычное явление: **если в JavaScript написать `1/0`, то результатом будет `Inf`, а вот `1/-0` вернёт `-Inf`.** Для чисел с плавающей запятой действительно определены два нуля: положительный и отрицательный! К счастью, в программе это обычно не нужно учитывать. Оба нуля при сравнении равны и на выводе они, в большинстве языков программирования, тоже выглядят одинаково. Знак нуля важен только для операций деления и умножения. Поэтому **во многих языках программирования нельзя даже определить константу со значением −0, она автоматически преобразуется в положительный ноль** (именно по этой причине пришлось использовать JavaScript в примере).

Как же быть с MySQL?

Типы данных **NUMERIC и DECIMAL реализованы в MySQL** как один и тот же тип - это разрешается стандартом SQL92. Они используются для величин, для которых важно сохранить повышенную точность, например **для денежных данных.** Требуемая точность данных и масштаб могут задаваться (и обычно задаются) при объявлении столбца данных одного из этих типов, например:

amount DECIMAL(5,2)

В этом примере - **5 (точность) представляет собой общее количество значащих десятичных знако**в, с которыми будет храниться данная величина, а цифра **2 (масштаб) задает количество десятичных знаков после запятой.** Следовательно, в этом случае интервал величин, которые могут храниться в столбце salary, составляет от -99,99 до 99,99 (в действительности для данного столбца MySQL обеспечивает возможность хранения чисел вплоть до 999,99, поскольку можно не хранить знак для положительных чисел).

Величины типов DECIMAL и NUMERIC хранятся как строки, а не как двоичные числа с плавающей точкой, чтобы сохранить точность представления этих величин в десятичном виде.