IEEE 754双精度浮点格式和JavaScript中的Number #19

Open xwcoder opened this issue on 10 Oct 2019 · 1 comment

xwcoder commented on 10 Oct 2019

[TOC]

这篇内容尝试解释清楚IEEE 754双精度浮点格式和JavaScript中的Number类型，并简单介绍Smi。

Number类型的定义

首先看一下ECMA-262对Number类型的定义：

> The Number type has exactly 18437736874454810627 (that is, 264 - 253 + 3) values, representing the double-precision 64-bit binary format IEEE 754-2008 values as specified in the IEEE Standard for Floating-Point Arithmetic, except that the 9007199254740990 (that is, 253 - 2) distinct "Not-a-Number" values of the IEEE Standard are represented in ECMAScript as a single special NaN value.

在 `BigInt` 被引入之前，JavaScript中只有Number一种数值类型，采用IEEE 754双精度浮点格式表示，不区分整型和浮点型。

目前(2019-05)， `BigInt` 处于Stage 3阶段，当前可以在v8中使用BigInt，Node.js中也引入了使用BigInt的API，比如 `process.hrtime.bigint()` 。

十进制到二进制转换

文中内容会涉及到相关计算，所以先再熟悉下十进制到二进制的转换计算。

> 整数部分，除2取余，直至商数为0，从下到上读余数，即是二进制的整数部分。小数部分，用其乘2，取其整数部分的结果，再用计算后的小数部分依此重复计算，算到小数部分全为0为止，从上到下读所有计算后整数部分的数字，即是二进制的小数部分。-- wikipedia

举例，将$59.25_{(10)}$转换为二进制：

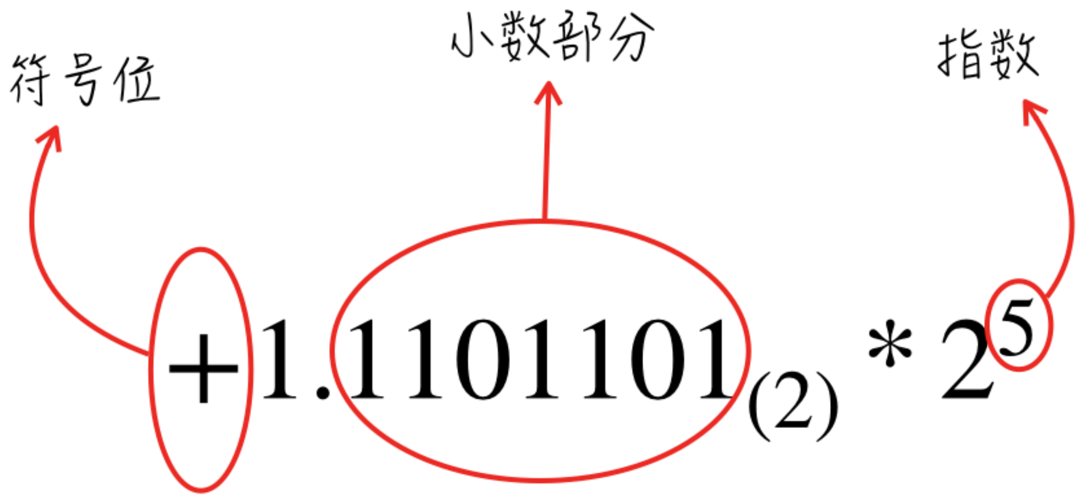
```
// 整数部分:
59 ÷ 2 = 29 ... 1
29 ÷ 2 = 14 ... 1
14 ÷ 2 =  7 ... 0
 7 ÷ 2 =  3 ... 1
 3 ÷ 2 =  1 ... 1
 1 ÷ 2 =  0 ... 1
// 小数部分:
0.25×2=0.5
0.50×2=1.0
```

$59.25_{(10)}$ = $111011.01_{(2)}$

不难发现，**对于小数部分最后一位是1，2，3，4，6，7，8，9的十进制数是不能转换成有限位数的二进制的。**

科学计数法

对于任何一个十进制数都可以用科学计数法表示，比如322000 = 3.22 x 10^5。同理，二进制数也可以用科学计数法表示，比如$59.25_{(10)}$ = +$111011.01_{(2)}$ = +$11.101101_{(2)}$ x 2^4 = +$1.1101101_{(2)}$ x 2^5。



整数部分总是可以精确到1，那么只需要记录**符号位、小数、指数位**这三个部分的值就可以完整表示一个二进制数。

IEEE 754双精度浮点格式

IEEE 754

IEEE标准协会（英文Institute of Electrical and Electronics Engineers Standards Association，简称IEEE-SA）是电气和电子工程师协会（IEEE）下辖的标准制定机构，其标准制定内容涵盖信息技术、通信、电力和能源等多个领域，已制定了900多个现行工业标准。

其中IEEE 754是二进制浮点数算术标准。标准中定义了二进制浮点数的格式，其中包括双精度浮点格式。

Name	Common name	Base	Significand bits[b] or digits	Decimal digits	Exponent bits
binary16	Half precision	2	11	3.31	5
binary32	Single precision	2	24	7.22	8
binary64	Double precision	2	53	15.95	11
binary128	Quadruple precision	2	113	34.02	15
binary256	Octuple precision	2	237	71.34	19
decimal32		10	7	7	7.58
decimal64		10	16	16	9.58
decimal128		10	34	34	13.58

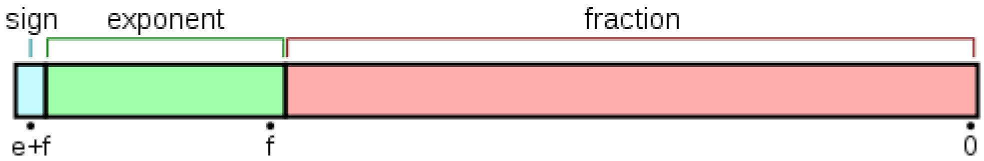
单精度浮点格式使用32位(4字节)表示，也被称为 `binary32` 。双精度浮点格式使用64位(8字节)表示，也被称为 `binary64` 。

双精度浮点格式

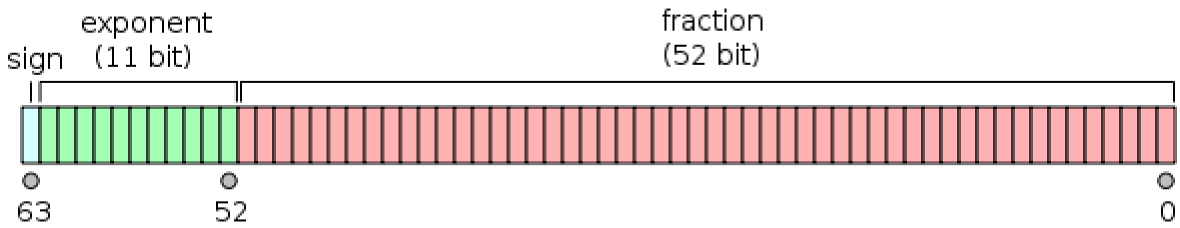
IEEE 754二进制浮点格式有**规约**和**非规约**两种形式。这部分内容主要介绍双精度浮点格式，先介绍其规约形式，之后介绍非规约形式，最后介绍**特殊值**。

其他精度的浮点格式表示方法类似，只是在**各部分比特位位数**、**指数偏移量**等方面有差异。

规约形式



在**科学计数法**部分介绍过，完整表示一个二进制数只需要记录**符号**、**指数位**、**小数**三部分信息。IEEE 754浮点格式就是按照**科学计数法**的方式存储值的。不同精度格式的指数和小数部分有不同的位数。



具体到双精度浮点格式，各部分的位数如下：

- 符号位(sign bit): 1位。
- 指数位(exponent bit): 11位。
- 小数部分(fraction bit): 52位。

符号位

符号位很好理解，1代表负，0代表正。

指数位

指数位有11位。指数部分的值e使用如下运算规则得到：

1. 将指数部分按11位无符号整数解析得到e1，所以e1的取值范围是[0, 2047]。
2. 其中0 `(00000000000)` 和2047 `(11111111111)` 有特殊含义，另作他用；所以e1的取值范围是[1, 2046]，即编码范围 `[00000000001, 11111111110]` 。 `00000000000` 和 `11111111111` 会在**非规约形式**和**特殊值**中用到，后面会有介绍。
3. `e = e1 - 1023` ，所以e的取值范围是[-1022, 1023]。减去的1023被称为**指数偏移量**，不同精度的指数偏移量不同，双精度浮点格式指数偏移量是1023，单精度浮点格式指数偏移量是127。

关于规则3中 `e1 - 1023` 以及不同精度的指数偏移量定义在这里：

> When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.

> - For a single-precision number, the exponent is stored in the range 1 .. 254 (0 and 255 have special meanings), and is interpreted by subtracting the bias for an 8-bit exponent (127) to get an exponent value in the range −126 .. +127.
> - For a double-precision number, the exponent is stored in the range 1 .. 2046 (0 and 2047 have special meanings), and is interpreted by subtracting the bias for an 11-bit exponent (1023) to get an exponent value in the range −1022 .. +1023.
> - For a quad-precision number, the exponent is stored in the range 1 .. 32766 (0 and 32767 have special meanings), and is interpreted by subtracting the bias for a 15-bit exponent (16383) to get an exponent value in the range −16382 .. +16383.

计算举例：
10000000101_2 = 1029_{10}

e = 1029 - 1023 = 6

小数位

小数部分有52位。**整数部分总是1**，不用存储。

所以有效数字位数共53位：52小数位数 + 1位整数位数。

$log2^{53}\approx15.95$

所以双精度浮点数可以保证15位十进制有效数。

计算举例

`0 10000000011 0111000000000000000000000000000000000000000000000000`

符号位为0，即+。
指数位e = `10000000011`(1027) - 1023 = 4。
有效数为 1.0111。

即+$1.0111 * 2^4$ = 23

规约形式的最小正数是：
`0 00000000001 0000000000000000000000000000000000000000000000000001`

```
> (1 + Math.pow(2, -52)) * Math.pow(2, -1022)
<· 2.225073858507202e-308
```

非规约形式

当指数位编码是`00000000000`，并且小数部分不为0时为**非规约形式**。与规约形式相比有两点不同：

1. 指数部分的偏移量比规约形式少1，对双精度浮点格式来说即1022，所以非规约形式的指数e总是-1022。
2. **整数部分为0**。

特殊值

- 指数部分编码为 `11111111111`，小数部分为0时，表示正负无穷。
- 指数部分编码为 `11111111111`，小数部分不为0时，表示NaN。
- 指数部分和小数部分编码全为0时，表示±0。

```
0 11111111111 0000000000000000000000000000000000000000000000000000 // +∞
1 11111111111 0000000000000000000000000000000000000000000000000000 // -∞
0 11111111111 0000000000000000000000000000000000000000000000000001 // NaN
0 11111111111 1000000000000000000000000000000000000000000000000001 // NaN
0 00000000000 0000000000000000000000000000000000000000000000000000 // +0
1 00000000000 0000000000000000000000000000000000000000000000000000 // -0
```

```
> Math.pow(2, 1024)
< Infinity
> Number.POSITIVE_INFINITY
< Infinity
> Math.pow(2, 1024) === Number.POSITIVE_INFINITY
< true
> -Math.pow(2, 1024) === Number.NEGATIVE_INFINITY
< true
```

Number

这部分主要计算Number类型上定义的几个常量值。

Number.MAX_VALUE

二进制表示如下，即指数部分和小数部分均取最大值。
`0 11111111110 1111111111111111111111111111111111111111111111111111`

$(2 - 2^{-52}) * 2^{1023}$

```
> (2 - Math.pow(2, -52)) * Math.pow(2, 1023)
< 1.7976931348623157e+308
> Number.MAX_VALUE
< 1.7976931348623157e+308
```

Number.MIN_VALUE

在使用IEEE 754-2008双精度浮点格式的实现中，`Number.MIN_VALUE` 表示**非规约形式**的最小正数。

> The value of Number.MIN_VALUE is the smallest positive value of the Number type, which is approximately 5 × 10-324.
> In the IEEE 754-2008 double precision binary representation, the smallest possible value is a denormalized number. If an implementation does not support denormalized values, the value of Number.MIN_VALUE must be the smallest non-zero positive value that can actually be represented by the implementation. -- ecma262

非规约形式下的最小正数：
`0 00000000000 0000000000000000000000000000000000000000000000000001`
$2^{-52} * 2^{-1022}$

```
> Math.pow(2, -52) * Math.pow(2, -1022)
< 5e-324
> Number.MIN_VALUE
< 5e-324
```

Number.MAX_SAFE_INTEGER/Number.MIN_SAFE_INTEGER

`Number.MIN_SAFE_INTEGER = -Number.MAX_SAFE_INTEGER`

由于双精度浮点格式的小数部分只有52位，所以其能准确表示的最大正整数为：符号位为0，小数部分编码全部为1，指数位计算结果为52。即$1.111...111 * 2^{52} = 2^{53} - 1$ 。超过此数值的整数无法使用双精度浮点格式准确表示。

```
> Number.MAX_SAFE_INTEGER
< 9007199254740991
> Math.pow(2, 53) - 1
< 9007199254740991
> Math.pow(2, 53)
< 9007199254740992
> Math.pow(2, 53) + 1
< 9007199254740992
> Math.pow(2, 53) + 2
< 9007199254740994
> Math.pow(2, 53) + 3
< 9007199254740996
```

因此，为了操作安全，数组在一些诸如`concat`，`from`等方法中要判断操作结果的长度是否在安全范围内。

22.1.3.1 Array.prototype.concat (...*arguments*)

When the `concat` method is called with zero or more arguments, it returns a
argument in order.

The following steps are taken:

1. Let *O* be ? ToObject(**this** value).
2. Let *A* be ? ArraySpeciesCreate(*O*, 0).
3. Let *n* be 0.
4. Let *items* be a List whose first element is *O* and whose subsequent eleme
5. Repeat, while *items* is not empty
   a. Remove the first element from *items* and let *E* be the value of the el
   b. Let *spreadable* be ? IsConcatSpreadable(*E*).
   c. If *spreadable* is **true**, then
      i. Let *k* be 0.
      ii. Let *len* be ? ToLength(? Get(*E*, `"length"`)).
      iii. If $n + len > 2^{53} - 1$, throw a **TypeError** exception.
      iv. Repeat, while $k < len$

Smi

在JavaScript引擎性能优化相关的文章中经常会看到Smi(Small Integer)，SMIs(Small Integers)，即小整数。这部分内容简单介绍Smi。

Smi是JavaScript引擎的优化手段，这部分内容主要以v8进行介绍，其他JS引擎也有类似优化。

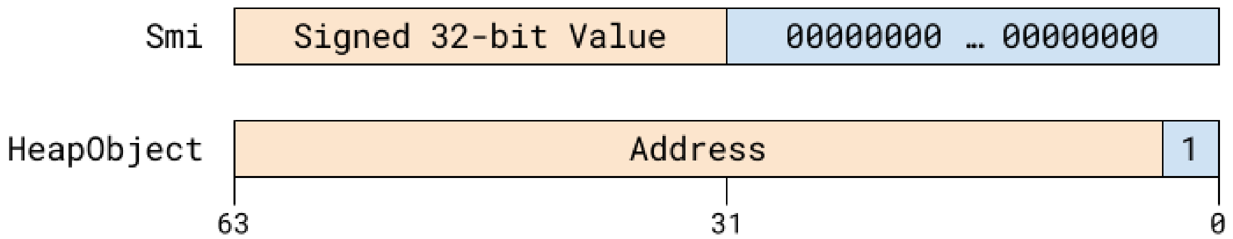
什么是Smi

通过前面的介绍我们知道BigInt出现之前，JavaScript中只有Number一种数值类型，采用IEEE 754双精度浮点格式表示，不区分整型和浮点型。但是在程序中会频繁使用小整数，比如数组的下标和数学运算等。如果总是从堆(heap)中分配内存存储数值并被gc管理，并且进行浮点运算，开销太大而且性能低。所以v8在内部对小整数(Smi)使用了整数格式表示，而不是IEEE 754浮点格式。

这样在v8内部就有两类值：一种是Smi，直接表示一个整数；一种是堆对象，称作HeapObject。
可以参考src/objects.h中的注释：

```
// Inheritance hierarchy:
// - Object
//   - Smi          (immediate small integer)
//   - HeapObject   (superclass for everything allocated in the heap)
//     - JSReceiver  (suitable for property access)
//       - JSObject
//         - JSArray
//         - JSArrayBuffer
```

```
// Formats of Object::ptr_:
//  Smi:        [31 bit signed int] 0
//  HeapObject: [32 bit direct pointer] (4 byte aligned) | 01
```



v8通过最低的一个比特位来区分Smi和指向HeapObject的指针：最低比特位为1时是指向HeapObject的指针，为0时是Smi。include/v8-internal.h。

```

// Tag information for HeapObject.
const int kHeapObjectTag = 1;
const int kWeakHeapObjectTag = 3;
const int kHeapObjectTagSize = 2;
const intptr_t kHeapObjectTagMask = (1 << kHeapObjectTagSize) - 1;

// Tag information for Smi.
const int kSmiTag = 0;
const int kSmiTagSize = 1;
const intptr_t kSmiTagMask = (1 << kSmiTagSize) - 1;
```

Smi的范围

在64位系统中，Smi的低32位全部为0，只使用高32位表示数值，所以其取值范围是[-2^{31}, $2^{31} - 1$]

。

在32位系统中，Smi的最低一位为0，使用剩余的31位表示数值，所以其取值范围是[-2^{30}, $2^{30} - 1$]

。

Smi的范围信息也定义在include/v8-internal.h。

```
const int kSmiShiftSize = PlatformSmiTagging::kSmiShiftSize;
const int kSmiValueSize = PlatformSmiTagging::kSmiValueSize;
const int kSmiMinValue = (static_cast<unsigned int>(-1)) << (kSmiValueSize - 1);
const int kSmiMaxValue = -(kSmiMinValue + 1);
constexpr bool SmiValuesAre31Bits() { return kSmiValueSize == 31; }
constexpr bool SmiValuesAre32Bits() { return kSmiValueSize == 32; }

```

64位系统中， `kSmiValueSize` 是32；32位系统中 `kSmiValueSize` 是31。

```
// 以64位系统为例
kSmiMinValue = (static_cast<unsigned int>(-1)) << (kSmiValueSize - 1)
  = 111...111 << (32 - 1)
  = 111...111 << 31 // -1的补码
  = 111...111000...000 // 31个0
  = -2^31
kSmiMaxValue = -(kSmiMinValue + 1) = 2^31 - 1
```

参考资料

- wikipedia: Double-precision floating-point format
- wikipedia: IEEE 754
- ecma 262
- An Introduction to Speculative Optimization in V8, by Meurer