JS 中的 number 类型为何如此怪异？

对于 JavaScript 开发者来说，或多或少都遇到过 `js` 在处理数字上的奇怪现象，比如：

```
> 0.1 + 0.2
0.30000000000000004

> 0.1 + 1 - 1
0.10000000000000009

> 0.1 * 0.2
0.020000000000000004

> Math.pow(2, 53)
9007199254740992

> Math.pow(2, 53) + 1
9007199254740992

> Math.pow(2, 53) + 3
9007199254740996
```

如果想要弄明白为什么会出现这些奇怪现象，首先要弄清楚 JavaScript 是怎样编码数字的。

1. JavaScript 是怎样编码数字的

JavaScript 中的数字，不管是整数、小数、分数，还是正数、负数，全部是浮点数，都是用 8 个字节（64 位）来存储的。

一个数字（如 `12`、 `0.12`、 `-999`）在内存中占用 8 个字节（64 位），存储方式如下：

1. `0-51`：分数部分（52 位）
2. `52-62`：指数部分（11 位）
3. `63`：符号位（1 位：0 表示这个数是正数，1 表示这个数是负数）

符号位很好理解，用于指明是正数还是负数，且只有 1 位、两种情况（0 表示正数，1 表示负数）。

其他两部分是分数部分和指数部分，用于计算一个数的绝对值。

1.1 绝对值计算公式

```
1: abs = 1.f * 2 ^ (e - 1023)                0 < e < 2047
2: abs = 0.f * 2 ^ (e - 1022)                e = 0, f > 0
3: abs = 0                                          e = 0, f = 0
4: abs = NaN                                        e = 2047, f > 0
5: abs = ∞ (infinity, 无穷大)                  e = 2047, f = 0
```

说明：

- 这个公式是二进制的算法公式，结果用 `abs` 表示，分数部分用 `f` 表示，指数部分用 `e` 表示
- `2^(e-1023)` 表示 `2` 的 `e-1023` 次方
- 因为分数部分占 52 位，所以 `f` 的取值范围为 `00...00`（中间省略 48 个 0）到 `11...11`（中间省略 48 个 1）
- 因为指数部分占 11 位，所以 `e` 的取值范围为 `0`（`00000000000`） 到 `2047`（`11111111111`）

从上面的公式可以看出：

- `1` 的存储方式：`1.00*2^(1023-1023)`（`f=0000...,e=1023`，`...` 表示 48 个 0）
- `2` 的存储方式：`1.00*2^(1024-1023)`（`f=0000...,e=1024`，`...` 表示 48 个 0）
- `9` 的存储方式：`1.01*2^(1025-1023)`（`f=0100...,e=1025`，`...` 表示 48 个 0）

- 0.5 的存储方式：1.00*2^(1022-1023)（f=0000...,e=1022，... 表示 48 个 0）
- 0.625 的存储方式：1.01*2^(1021-1023)（f=0100...,e=1021，... 表示 48 个 0）

1.2 绝对值的取值范围与边界

从上面的公式可以看出：

1.2.1 0<e<2047

当 0<e<2047 时，取值范围为：f=0,e=1 到 f=11...11,e=2046（中间省略 48 个 1）

即：Math.pow(2,-1022) 到 ~=Math.pow(2,1024)-1（~= 表示约等于）

这当中，~=Math.pow(2,1024)-1 就是 Number.MAX_VALUE 的值，js 所能表示的最大数值。

1.2.2 e=0,f>0

当 e=0,f>0 时，取值范围为：f=00...01,e=0（中间省略 48 个 0）到 f=11...11,e=0（中间省略 48 个 1）

即：Math.pow(2,-1074) 到 ~=Math.pow(2,-1022)（~= 表示约等于）

这当中，Math.pow(2,-1074) 就是 Number.MIN_VALUE 的值，js 所能表示的最小数值（绝对值）。

1.2.3 e=0,f=0

这只表示一个值 0，但加上符号位，所以有 +0 与 -0。

但在运算中：

```
> +0 === -0
true
```

1.2.4 e=2047,f>0

这只表示一种值 NaN。

但在运算中：

```
> NaN == NaN
false

> NaN === NaN
false
```

1.2.5 e=2047,f=0

这只表示一个值 ∞ (infinity，无穷大)。

在运算中：

```
> Infinity === Infinity
true

> -Infinity === -Infinity
true
```

1.3 绝对值的最大安全值

从上面可以看出，8 个字节能存储的最大数值是 Number.MAX_VALUE 的值，也就是 ~=Math.pow(2,1024)-1。

但这个数值并不安全：从 1 到 Number.MAX_VALUE 中间的数字并不连续，而是离散的。

比如： Number.MAX_VALUE-1,Number.MAX_VALUE-2 等数值都无法用公式得出，就存储不了。

所以这里引出了最大安全值 Number.MAX_SAFE_INTEGER，也就是从 1 到 Number.MAX_SAFE_INTEGER 中间的数字都是连续的，处在这个范围内的数值计算都是安全的。

当 f=11...11,e=1075（中间省略 48 个 1）时，取得这个值 111...11（中间省略 48 个 1），即 Math.pow(2,53)-1。

大于 Number.MAX_SAFE_INTEGER：Math.pow(2,53)-1 的数值都是离散的。

比如： Math.pow(2,53)+1, Math.pow(2,53)+3 不能用公式得出，无法存储在内存中。

所以才会有文章开头的现象：

```
> Math.pow(2, 53)
9007199254740992

> Math.pow(2, 53) + 1
9007199254740992

> Math.pow(2, 53) + 3
9007199254740996
```

因为 Math.pow(2,53)+1 不能用公式得出，就无法存储在内存中，所以只有取最靠近这个数的、能够用公式得出的其他数， Math.pow(2,53)，然后存储在内存中，这就是失真，即不安全。

1.4 小数的存储方式与计算

小数中，除了满足 m/(2^n)（ m,n 都是整数）的小数可以用完整的 2 进制表示之外，其他的都不能用完整的 2 进制表示，只能无限的逼近一个 2 进制小数（注： [2] 表示二进制， ^ 表示 N 次方）。

```
0.5 = 1 / 2 = [2]0.1
0.875 = 7 / 8 = 1 / 2 + 1 / 4 + 1 / 8 = [2]0.111
# 0.3 的逼近

0.25 ([2]0.01) < 0.3 < 0.5 ([2]0.10)

0.296875 ([2]0.0100110) < 0.3 < 0.3046875 ([2]0.0100111)

0.2998046875 ([2]0.01001100110) < 0.3 < 0.30029296875 ([2]0.01001100111)

```

```
... 根据公式计算，直到把分数部分的 52 位填满，然后取最靠近的数

0.3 的存储方式：[2]0.0100110011001100110011001100110011001100110011001100110011
(f = 0011001100110011001100110011001100110011001100110011001100110011, e = 1021)
```

从上面可以看出，小数中大部分都只是近似值，只有少部分是真实值，所以只有这少部分的值（满足 `m/(2^n)` 的小数）可以直接比较大小，其他的都不能直接比较。

```
> 0.5 + 0.125 === 0.625
true

> 0.1 + 0.2 === 0.3
false
```

为了安全的比较两个小数，引入 `Number.EPSILON[Math.pow(2,-52)]` 来比较浮点数。

```
> Math.abs(0.1 + 0.2 - 0.3) < Number.EPSILON
true
```

1.5 小数最大保留位数

`js` 从内存中读取一个数时，最大保留 `17` 位有效数字。

```
> 0.0100110011001100110011001100110011001100110011001100110011
0.30000000000000000
0.3
> 0.0100110011001100110011001100110011001100110011001100110010
0.29999999999999993
> 0.0100110011001100110011001100110011001100110011001100110100
0.30000000000000004
> 0.0000010100011110101110000101000111101011100001010001111100
0.020000000000000004
```

2. Number 对象中的常量

2.1 Number.EPSILON

表示 1 与 Number 可表示的大于 1 的最小的浮点数之间的差值。

```
Math.pow(2, -52)
```

用于浮点数之间安全的比较大小。

2.2 Number.MAXSAFEINTEGER

绝对值的最大安全值。

```
Math.pow(2, 53) - 1
```

2.3 Number.MAX_VALUE

`js` 所能表示的最大数值（8 个字节能存储的最大数值）。

```
~= Math.pow(2, 1024) - 1
```

2.4 Number.MINSAFEINTEGER

最小安全值（包括符号）。

```
-(Math.pow(2, 53) - 1)
```

2.5 Number.MIN_VALUE

`js` 所能表示的最小数值（绝对值）。

```
Math.pow(2, -1074)
```

2.6 Number.NEGATIVE_INFINITY

负无穷大。

```
-Infinity
```

2.7 Number.POSITIVE_INFINITY

正无穷大。

```
+Infinity
```

2.8 Number.NaN

非数字。

3. 寻找奇怪现象的原因

3.1 为什么 `0.1+0.2` 结果是 `0.30000000000000004`

与 `0.3` 的逼近算法类似。

```
0.1 的存储方式：[2]0.0001100110011001100110011001100110011001100110011001100110011010

(f = 1001100110011001100110011001100110011001100110011001100110011010, e = 1019)

0.2 的存储方式：[2]0.0011001100110011001100110011001100110011001100110011001100110011010

(f = 1001100110011001100110011001100110011001100110011001100110011010, e = 1020)
0.1 + 0.2: 0.0100110011001100110011001100110011001100110011001100110011001100111

(f = 0011001100110011001100110011001100110011001100110011001100110011001100111, e = 1021)
```

但 `f=0011001100110011001100110011001100110011001100110011001100110011001100111` 有 53 位，超过了正常的 52 位，无法存储，所以取最近的数：

```
0.1 + 0.2: 0.0100110011001100110011001100110011001100110011001100110011001100100

(f = 0011001100110011001100110011001100110011001100110011001100110011001100100, e = 1021)
```

`js` 读取这个数字为 `0.30000000000000004`

3.2 为什么 `Math.pow(2,53)+1` 结果是 `Math.pow(2,53)`

因为 `Math.pow(2,53)+1` 不能用公式得出，无法存储在内存中，所以只有取最靠近这个数的、能够用公式得出的其他数。

比这个数小的、最靠近的数：

```
Math.pow(2, 53)

(f = 00000000000000000000000000000000000000000000000000000000, e = 1076)
```

比这个数大的、最靠近的数：

```
Math.pow(2, 53) + 2

(f = 00000000000000000000000000000000000000000000000000000001, e = 1076)
```

取第一个数： `Math.pow(2,53)`。

所以：

```
> Math.pow(2, 53) + 1 === Math.pow(2, 53)
```

```
2.  true
```