

King of Dumbs

笨蛋之王

级别: ♠ 技术

为什么电脑说 $0.1 + 0.2$ 不等于 0.3 ?

0.1 在二进制里写不完，电脑只好把它舍入。什么是浮点数，误差为什么会累积，AI 和记账又该用什么数。分三个层次讲，配一个互动小工具。

中文 发布于 2026年10月15日 更新于 2026年10月16日



扫码打开可互动的课程

<https://kingofdumbs.com/zh/lessons/floating-point>

你将学到

- 为什么 0.1 无法在二进制中精确存储
- 浮点数：位数固定的二进制科学记数法
- 舍入误差如何累积，以及如何带容差比较
- AI 里的 float32、float16 和 bfloat16，以及记账用的十进制

学这一课之前

如果有什么看不懂，这些课会从零讲起：

1. [7×10²⁷ 到底有多大？轻松搞懂超大数字](#)

本课会出现的符号

Δ 变化量：变了多少 (♠) · E_a 下标：右下角的标签 (♠) · Σ 求和：全部加起来 (♠) · x_i 下标 i：第 i 个元素 (♠) · $\approx$ 约等于 (♠) · $\leq \geq$ 小于等于 / 大于等于 (♠) · $|x|$ 绝对值：去掉正负号 (♠) · 10^{-3} 负指数：表示除 (♦♠) · $\log$ 对数：要乘几次？ (♠) · $\lfloor x \rfloor$ 向下取整 (♠) · ϵ epsilon：很小的量 (♠) · δ 小写 delta：小误差 (♠) · $\in$ 属于 (♠)

在电脑上打开浏览器，按 F12 打开控制台，输入 $0.1 + 0.2$ ，回车。答案是 0.30000000000000004 。换成 Python 试试：一样。一台专门用来计算的机器，竟然算错了一道小学加法。

这一课的傻问题是：**为什么电脑说 $0.1 + 0.2$ 不等于 0.3 ？** 它没坏。它存数字的方式决定了 0.1 存不准。这一课讲清楚这种方式是什么，为什么几乎察觉不到，以及什么时候会露馅。

选一张牌：零基础选 ♣，不怕简单公式选 ♦，想看记号选 ♠。随时可以换。

0.1 在二进制里放不下

♦ → ♠ 从进阶到技术

♦ 里的说法	♠ 里的写法
二进制	以 2 为底； $(\cdot)_2$ 表示用二进制写的数
0011 这一组永远重复下去	$0.0(0011)_2 = \sum_{k \geq 1} (2^{-4k} + 2^{-4k-1})$ ，循环展开
这和十进制里的 1/3 是一回事	既约分数 $p/q \in \mathbb{Q}$ （有理数）：在二进制中展开有限 $\Leftrightarrow q = 2^k$
53 位二进制数字	精度 $p = 53$ 位
舍入到最接近的数	$\text{fl}(x)$ ：就近舍入，平局时取偶
相加	$x \oplus y = \text{fl}(x + y)$ ，浮点加法
下一个数	相差一个 ulp（unit in the last place，末位单位）的邻居，这里是 2^{-54}

$$0.1 = 0.0(0011)_2 = \sum_k (2^{-4k} + 2^{-4k-1})$$

$$p/q \in \mathbb{Q} \text{ 在二进制中有限} \Leftrightarrow q = 2^k$$

$$\text{fl}(0.1) = 3602879701896397 / 2^{55}$$

$$\text{fl}(0.1) - 0.1 = 1/(5 \cdot 2^{55}) \approx 5.55 \times 10^{-18}$$

$$0.1 \oplus 0.2 = \text{fl}(0.3) + 2^{-54}$$

既约分数 p/q 在二进制中展开有限，当且仅当 q 是 2 的幂。1/10 的分母里有因子 5，所以是循环小数：
 $0.1 = 0.0(0011)_2 = \sum_{k \geq 1} (2^{-4k} + 2^{-4k-1})$ 。用几何级数验证： $(3/32) / (15/16) = 1/10$ 。

当 $p = 53$ 时， $\text{fl}(0.1)$ 是最接近的分数 $J/2^N$ ，其中 J 有 53 位： $J = 7205759403792794$ ， $N = 56$ ，约分后是 $3602879701896397 / 2^{55}$ （Python 教程）。因为 $10 \times 3602879701896397 = 36028797018963970$ ，而 $2^{55} = 36028797018963968$ ：

$$\text{fl}(0.1) - 0.1 = 2 / (10 \cdot 2^{55}) = 1 / (5 \cdot 2^{55}) \approx 5.55 \times 10^{-18}$$

也就是向上舍入了。在 $0.1 \oplus 0.2$ 中，精确和 $\text{fl}(0.1) + \text{fl}(0.2)$ 恰好落在两个 53 位数的正中间。按“平局取偶”的规则，它被送到上面那个数，比 $\text{fl}(0.3)$ 大一个 ulp，即 $2^{-54} \approx 5.55 \times 10^{-17}$ 。所以 $0.1 + 0.2 == 0.3$ 在 Python、JavaScript 以及任何使用这种格式的语言里都是假的（用精确算术自行计算；在

JavaScript 里, $0.1 + 0.2 - 0.3$ 得到 2^{-54})。

笨蛋之王说:

电脑做加法没算错。问题是它手里的数根本不是 0.1: 放不下。



浮点数：二进制的科学记数法

◆→♠ 从进阶到技术

◆里的说法	♠里的写法
符号	$s \in \{0, 1\}$
指数	E, 11 位, 偏移量 1023: 实际指数是 $E - 1023$
尾数	$(1.f)_2$, f 是存下来的 52 位, 1 是隐含的
binary64	IEEE 754-2019 的格式, $p = 53$, $e_{min} = -1022$, $e_{max} = 1023$
相邻两个数之间的间隔	$ulp(x) = 2^{(\lfloor \log_2 x \rfloor - 52)}$, $\lfloor \cdot \rfloor$ 表示向下取整
就舍入到最近的那个	$fl(x) = x(1 + \delta)$, $ \delta \leq u = 2^{-53}$ (u: 单位舍入误差)
紧挨着 1 的上方, 间隔是 2.2×10^{-16}	$\epsilon = ulp(1) = 2^{-52}$ (Number.EPSILON、sys.float_info.epsilon)
无穷大和没有意义的结果	$\pm\infty$ 和 NaN (not a number, 非数), E = 2047; E = 0 用于零和次正规数

$$x = (-1)^s \cdot (1.f)_2 \cdot 2^{(E - 1023)}$$

$$ulp(x) = 2^{(\lfloor \log_2 |x| \rfloor - 52)}$$

$$fl(x) = x(1 + \delta), \quad |\delta| \leq u = 2^{-53}$$

$$\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$$

$$2^{53} \approx 9 \times 10^{15}$$

一个正规的 binary64 等于 $x = (-1)^s \cdot (1.f)_2 \cdot 2^{(E - 1023)}$ ，其中 $1 \leq E \leq 2046$ 。E = 0 和 E = 2047 留给零、次正规数（没有隐含的 1）、 $\pm\infty$ 和 NaN。参数 p = 53、emin = -1022、emax = 1023 来自 Goldberg（1991）的表 D-1；这项标准目前的版本是 IEEE 754-2019。

在 2^e 和 2^{e+1} 之间有 2^{52} 个等距的数，间隔为 $ulp(x) = 2^{(\lfloor \log_2 |x| \rfloor - 52)}$ 。就近舍入时， $fl(x) = x(1 + \delta)$ ， $|\delta| \leq u = 2^{-53}$ ，每一次基本运算也满足同样的界： $x \oplus y = (x + y)(1 + \delta)$ （Goldberg）。

注意名称。MDN、Python 和 PyTorch 把 $\epsilon = 2^{-52} = ulp(1)$ ，也就是 1 到下一个数的距离，叫作 epsilon。Goldberg 所说的 *machine epsilon* 是相对误差的上界，只有它的一半： $u = 2^{-53}$ 。整数在 $2^{53} = 9007199254740992$ 以内是精确的；超过它， $ulp \geq 2$ （MDN，`Number.MAX_SAFE_INTEGER = 253 - 1`）。



舍入误差会累积

◆→♠从进阶到技术

◆里的说法	♠里的写法
每次运算都要舍入	$x \oplus y = fl(x + y)$ ， $x \ominus y = fl(x - y)$
误差会一点点累积	递归求和 $s \leftarrow s \oplus x_j$ ：第 j 项的扰动最多为 $(n - j)u$
顺序会改变结果	$\oplus$ 满足交换律，但不满足结合律
带容差比较	$ a - b \leq \max(rel_tol \cdot \max(a , b), abs_tol)$ ， <code>rel_tol = 10⁻⁹</code> <code>(math.isclose)</code>
小心地求和	Kahan 补偿求和，c 是每一步带着的误差
在内部补偿误差	Neumaier 求和，Kahan 求和的一种变体（gh-100425）
只舍入一次	最后只舍入一次（ <code>math.fsum</code> ）； $O(nu^2)$ ： nu^2 量级或更小的项

$s \leftarrow s \oplus x_j : |\delta_j| < (n - j)u$ (朴素求和)

Kahan: $y = x_j \ominus c; t = s \oplus y$

$c = (t \ominus s) \ominus y; s = t$

$|\text{error}| \leq (2u + O(nu^2)) \sum |x_j|$

$(10^{30} \oplus -10^{30}) \oplus 1 = 1$

$10^{30} \oplus (-10^{30} \oplus 1) = 0$

递归求和时，每一项 x_j 都带着一个因子 $(1 + \delta_j)$ ， $|\delta_j| < (n - j)u$ ：误差随 n 增长 (Goldberg, 1991；他把 $u = 2^{-53}$ 叫作 epsilon，见上一部分)。Kahan 补偿求和把每次 $\oplus$ 丢掉的部分存在 c 里，在下一步还回去。它的误差保持在 $(2u + O(nu^2)) \sum |x_j|$ ，几乎与 n 无关 (Goldberg, 定理 8；Kahan, CACM 8, 40, 1965)。

没有结合律，顺序就很重要： $(10^{30} \oplus -10^{30}) \oplus 1 = 1$ ，但 $10^{30} \oplus (-10^{30} \oplus 1) = 0$ (Goldberg)。由此推出：把一个很长的求和分给几个处理器去算，会改变顺序，结果的最后几位可能不同。

比较时，`math.isclose(a, b)` 检查 $|a - b| \leq \max(\text{rel_tol} \cdot \max(|a|, |b|), \text{abs_tol})$ ，默认 $\text{rel_tol} = 10^{-9}$ ：大约 9 位一致 (Python 文档)。从 Python 3.12 起，`sum()` 使用 Neumaier 求和 (gh-100425)，`math.fsum` 只在最后舍入一次。

笨蛋之王说：

永远别问一个带小数的数是否等于另一个数。要问它们是否足够接近。

AI 用更少的位，记账用十进制

♦→♠ 从进阶到技术

♦里的说法	♠里的写法
约 7 位有效数字	$p \cdot \log_{10} 2$ 位十进制数字: $p = 24$ 时为 7.22
不能超过 65504	最大有限值 $(2 - 2^{1-p}) \cdot 2^{\text{emax}}$, emax 为最大指数
范围和 float32 一样	同样是 8 位指数: $\text{emax} = 127$
混合精度	<code>mixed precision</code> : 乘法用 FP16 或 BF16 (<code>float16</code> 、 <code>bfloat16</code>); 主权重和累加用 FP32 (<code>float32</code>)
网络权重	权重 w ; 梯度: 每个权重变动时, 损失 (网络的误差) 变化多少
谷歌	TPU (Tensor Processing Unit, 张量处理器): 谷歌的神经网络芯片
小的改变量就可能丢失	若 $ \Delta w < \text{ulp}(w)/2$, 则 $w \oplus \Delta w = w$
2048 倍	$2^{11} = 2^p$, $p = 11$
十进制算术	底数 $\beta = 10$ 的浮点数 (Python 的 <code>decimal</code> 模块)

	p	ϵ	max
binary64	53	2.2×10^{-16}	1.8×10^{308}
binary32	24	1.2×10^{-7}	3.4×10^{38}
binary16	11	9.8×10^{-4}	65504
bfloat16	8	7.8×10^{-3}	3.4×10^{38}

$$|\Delta w| < \text{ulp}(w)/2 \Rightarrow w \oplus \Delta w = w$$

精度为 p 位、最大指数为 emax 时, $\epsilon = 2^{1-p}$, 最大有限值为 $(2 - 2^{1-p}) \cdot 2^{\text{emax}}$ 。位的划分来自 NumPy、PyTorch 和谷歌; 表中的数值是自行计算的。对 binary16, $(2 - 2^{-10}) \cdot 2^{15} = 65504$, 与 Micikevicius 等 (2018) 给出的值相同。

只要 $|\Delta w| < \text{ulp}(w)/2$, 就有 $w \oplus \Delta w = w$; 在 FP16 中, 从 $|w| \geq 2^{11}|\Delta w|$ 起就可能发生 (Micikevicius 等)。他们的混合精度方案用 FP32 保存主权重, 用 FP16 做乘法, 并放大损失, 让小梯度不至于变成零: 在 FP16 里, 没有小于 2^{-24} 的正数; 在他们的目标检测网络 (SSD) 中, 放大 8 倍就够了。在谷歌的 TPU 上, 每次矩阵乘法都用 bfloat16 相乘、用 FP32 累加 (谷歌云, 2019)。小工具用 0.1 演示了这一点: 在 float16 里, 从 256 起 ulp 是 0.25, 而 $0.1 < 0.125$ 。

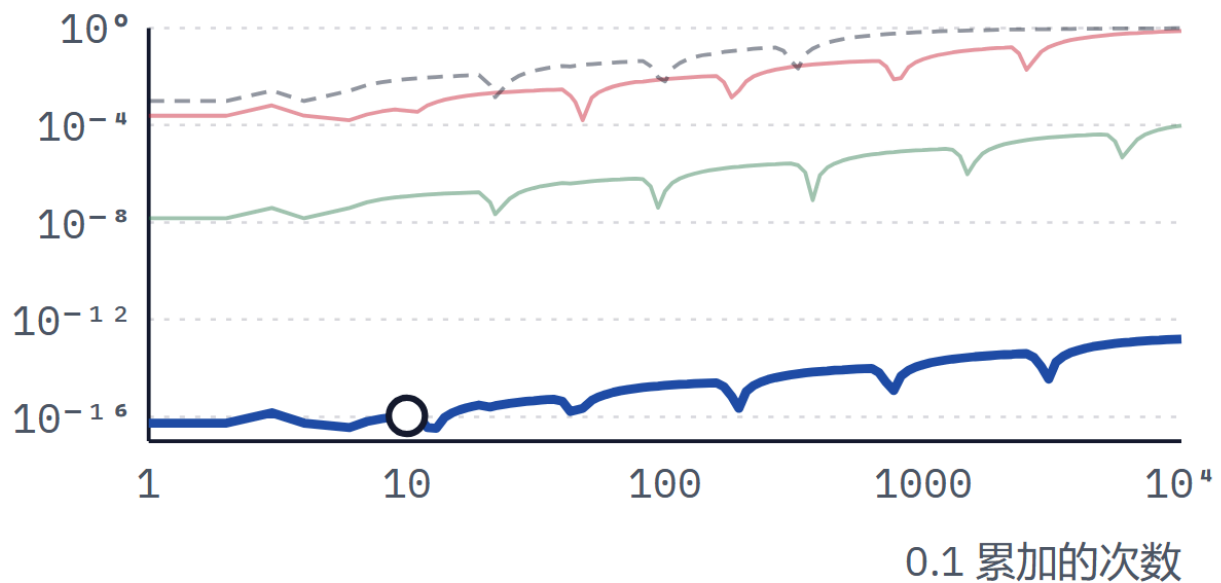
Python 的 decimal 模块是以 10 为底的浮点数: 0.1 能精确存储, $0.1 + 0.1 + 0.1 - 0.3$ 等于 0。记账也可以用整数记分, 在 binary64 中直到 2^{53} 都是精确的。



动手试试: 把 0.1 加了又加

选择电脑存数字的方式, 以及把 0.1 加多少次。比较实际得到的和应该得到的, 看看图上的误差怎么增长。试试 float16 加 5000 次。

相对误差



每种格式一条曲线；选中的那条加粗。

数字的存储方式

- ☒ — binary64, 最常用 (64 位)
- ☐ — float32 (32 位)
- ☐ — float16 (16 位)
- ☐ — bfloat16 (16 位, AI 常用)

0.1 累加的次数 10 次

应该得到 1

实际得到 **0.9999999999999999**

误差 1.1×10^{-16}

0.1 实际存成 0.10000000000000000055511151231257827021181583404541015625

偏差 1.1×10^{-16} : 相当于 9×10^{15} 分之一。



互动小工具。在网站上试一试: <https://kingofdumbs.com/zh/lessons/floating-point>



最后的挑战

1. 为什么电脑无法精确存储 0.1?

- a. 因为这个数太小了
- b. 因为 0.1 在二进制里有无限多位，就像十进制里的 1/3
- c. 因为程序写得不好

2. 你在循环里把 0.1 加十次，然后检查结果是否等于 1。会怎样？

- a. 是的，等于 1
- b. 不等于：结果是 0.9999999999999999，所以要带容差比较
- c. 程序会报错

3. 要存一个银行账户的余额，用什么最好？

- a. float16，省内存
- b. 用整数按分来记，或者用十进制算术
- c. binary64，它从来不舍入



三句话总结

1. 电脑用二进制计数，而 0.1 在二进制里没有尽头，就像十进制里的 1/3：它存下的是舍入后的值，稍微偏大一点。
2. 浮点数就是位数固定的二进制科学记数法（binary64 约 16 位）：每次运算都要舍入，误差会累积，求和的顺序也有影响。
3. 所以带小数的数要带容差比较；AI 用 16 位格式是因为省内存，钱则用整数按分记，或者用十进制。

参考资料

- 0.1 为什么存不准、它存下的值、`math.isclose` 和 `math.fsum`：Python 3.15, [教程第 15 章](#) 和 [math 模块](#)。
`sum()` 使用 Neumaier 求和：[Python 3.12 新特性](#)。十进制算术：[decimal 模块](#)。
- 表示、舍入、不满足结合律和 Kahan 求和：D. Goldberg, “What every computer scientist should know about floating-point arithmetic”, *ACM Computing Surveys* 23(1), 5–48 (1991), [doi:10.1145/103162.103163](#), [Oracle 重印版](#); W. Kahan, *Communications of the ACM* 8(1), 40 (1965), [doi:10.1145/363707.363723](#)。标准：[IEEE 754-2019](#)。
- 格式和位：[MDN, Number](#); [NumPy 2.5, 数据类型](#); [PyTorch 2.14, torch.dtype](#); [Excel 与浮点数 \(微软\)](#)。
- `bf16`：[谷歌云博客 \(2019\)](#) 和 [Cloud TPU 文档](#)。混合精度：P. Micikevicius et al., “Mixed Precision Training”, ICLR 2018, [arXiv:1710.03740](#)。

这一课是怎么做的： 在 Claude 的帮助下，根据上面的资料调研和撰写，并于 2026 年 10 月 15 日用第二个来源核对。各种格式下循环累加 0.1 的结果是在 JavaScript (Node 24) 里实测的，并有测试检查；没有单独链接的数字是根据资料计算的。插图是原创的。

答案

1. **b)** 因为 0.1 在二进制里有无限多位，就像十进制里的 $1/3$. 对。电脑按一半一半来数，十分之一永远拼不完。
2. **b)** 不等于：结果是 0.9999999999999999，所以要带容差比较. 很好。每次加法都会舍入，误差会累积。
3. **b)** 用整数按分来记，或者用十进制算术. 没错。和钱有关的，不要用二进制舍入。

King of Dumbs 的课程，kingofdumbs.com。这是离线版；可以动手玩的原版在：
<https://kingofdumbs.com/zh/lessons/floating-point>

更新于 2026年10月16日.