

King of Dumbs

笨蛋之王

级别: ♦ 进阶

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

每一格的值

$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$
0	0	0	1	1	0	0	1	1

九格: $\frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} = 0.099609375$

$0.1 = 0.0001100110011\dots_2$ (0011 不断重复)

电脑用**二进制**存数字，只有 0 和 1。在十进制里，小数点后每一位的值是前一位的十分之一；在二进制里，每一位的值是前一位的**一半**： $\frac{1}{2}$ 、 $\frac{1}{4}$ 、 $\frac{1}{8}$ 、 $\frac{1}{16}$

要把 0.1 写成二进制，就看哪些小块放得进去： $\frac{1}{2}$ 不行， $\frac{1}{4}$ 不行， $\frac{1}{8}$ 不行， $\frac{1}{16}$ 可以.....得到 0.0001100110011.....，**0011 这一组永远重复下去**。写九位能到 0.099609375，位数越多越接近，但永远写不完。**这和十进制里的 $\frac{1}{3}$ 是一回事。**

普通电脑保留 **53 位二进制数字**，并**舍入到最接近的数**。它的 0.1 精确地等于 0.1000000000000000055511151231257827021181583404541015625，比 0.1 稍大一点（Python 教程）。

0.2 和 0.3 也是这样，各有各的小误差。前两个数**相加**的结果，没有落在电脑用来表示 0.3 的那个数上，而是落在**下一个数**上：0.300000000000000004。

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

符号

指数 (11)

尾数 (52)



+ 2^{-4}

1.6

$0.1 = + 1.6 \times 2^{-4}$ · 约 16 位十进制有效数字

一个浮点数分三部分存储，就像科学记数法，只是底数是 2：

- **符号**，正或负；
- **指数**，表示乘以 2 的几次方；
- **尾数**，也就是那些数字。

最常用的格式叫 **binary64**（也叫“双精度”）：64 位，1 位符号，11 位指数，52 位尾数（MDN；NumPy）。尾数总是以 1 开头，这个 1 不用存，所以实际有 53 位二进制数字，约 16 位十进制数字。JavaScript 的数、Python 的 float 和 Excel 用的都是它。例如 $0.1 = + 1.6 \times 2^{-4}$ 。

因为位数总是一样多，**相邻两个数之间的间隔**会随数的大小变大。**紧挨着 1 的上方，间隔是 2.2×10^{-16}** 。在 10^{16} 附近，间隔是 2，所以 $10^{16} + 1$ 还是 10^{16} 。同样的原因，整数只在大约 9×10^{15} 以内是精确的（MDN）。

有些位组合留给了零、**无穷大和没有意义的结果**，比如 -1 的平方根（微软关于 Excel 的文章）。

运算结果落在两个能存储的数之间时，**就舍入到最近的那个**。每次舍入的误差最多是间隔的一半：不到 9×10^{15} 分之一。

舍入误差会累积

0.1 加 n 次

1 0.1	6 0.6
2 0.2	7 0.7
3 0.30000000000000004	8 0.7999999999999999
4 0.4	9 0.8999999999999999
5 0.5	10 0.9999999999999999

$10^{16} + 1 = 10^{16}$: 这个 1 放不进去

每次运算都要舍入，误差会一点点累积。在 JavaScript 里用循环累加 0.1:

次数	结果
3	0.30000000000000004
8	0.7999999999999999
10	0.9999999999999999
10000	1000.00000000001588

而且，**顺序会改变结果**。给一个很大的数加上一个小数，可能什么也没变 ($10^{16} + 1 = 10^{16}$)，所以 $(a + b) + c$ 不一定等于 $a + (b + c)$ (Goldberg, 1991)。

三个常用的办法:

- **带容差比较**，而不是用 `==`。在 Python 里，`math.isclose(0.1 + 0.1 + 0.1, 0.3)` 返回 `True` (Python 教程)。
- **小心地求和**。`math.fsum` 会记下丢掉的那些位，**只舍入一次**。从 Python 3.12 起，`sum()` 也会在**内部补偿误差**：在几乎所有安装版本中，`sum([0.1] * 10) == 1.0` 都返回 `True`，而手写的循环得到 `0.9999999999999999`。
- **显示时再舍入**，计算时不要舍入：价格保留两位小数。

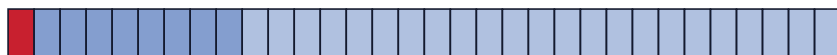
笨蛋之王说：

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。

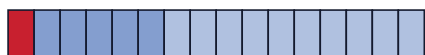


AI 用更少的位，记账用十进制

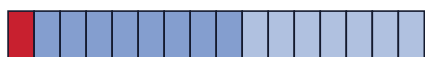
float32 ≈ 7 位有效数字 · 最大 3.4×10^{38}



float16 ≈ 3 位有效数字 · 最大 65504



bfloat16 ≈ 2 位有效数字 · 最大 3.4×10^{38}



除了 binary64，还有更小的格式：

- **float32**, 32 位 (1 + 8 + 23) : **约 7 位有效数字**。
- **float16**, 16 位 (1 + 5 + 10) : 约 3 位, 而且**不能超过 65504**。
- **bfloat16**, 16 位 (1 + 8 + 7) : 只有约 2 位, 但**范围和 float32 一样**。它是谷歌为机器学习设计的 (谷歌云博客, 2019) 。

神经网络大量使用这些小格式：内存只占一半，速度也可能更快。诀窍叫**混合精度**：乘法用 16 位做，同时用 32 位保存一份网络权重的副本。在 float16 里，如果被加的数比**一个小的改变量大 2048 倍**或更多，这个**小的改变量就可能丢失** (Micikevicius 等, 2018) 。又是那台地磅。

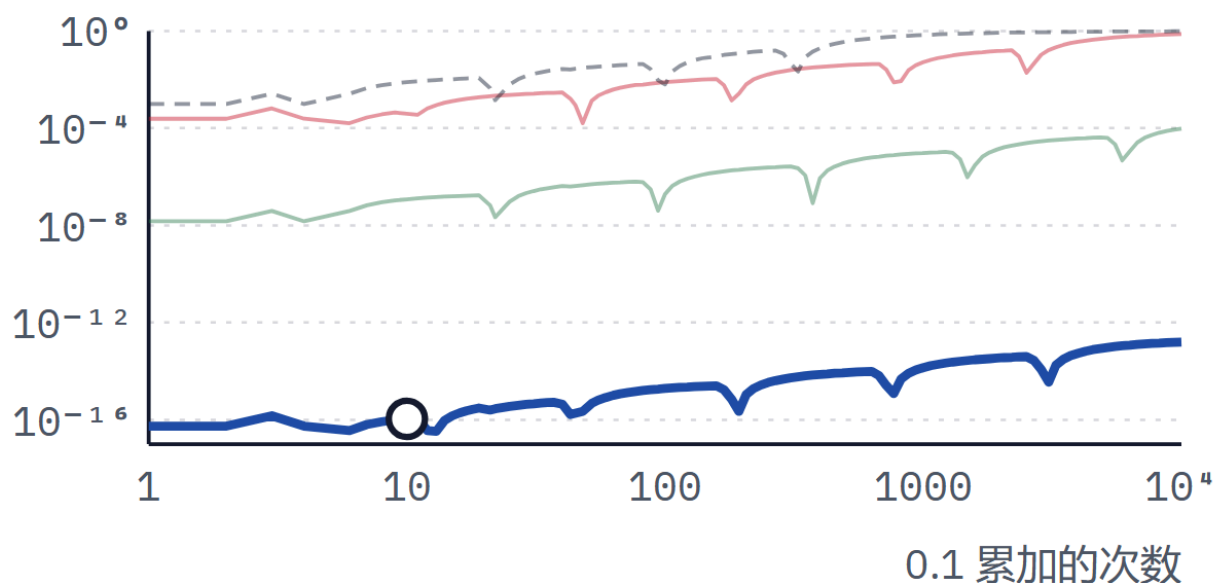
记账则相反：用**十进制算术**，像我们一样按十分之一、百分之一来数，或者用整数记分。在十进制里， $0.1 + 0.1 + 0.1 - 0.3$ 正好等于 0；在二进制里是 5.55×10^{-17} (Python decimal 模块的文档，这个模块就是为会计设计的) 。



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.