

Lecture 4: 知识 - 浮点数的表示与运算

浮点数的表示

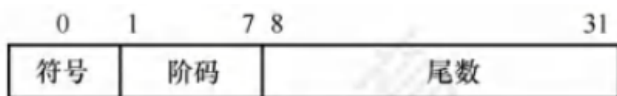
概念：浮点数表示法

- 根据适当的形式，将比例因子存储在数据中，使得小数点的位置可以根据需要而浮动；
- 即扩大了数的表示范围，又保持了数的有效精度；

浮点数的表示格式

概念：浮点数的表示格式

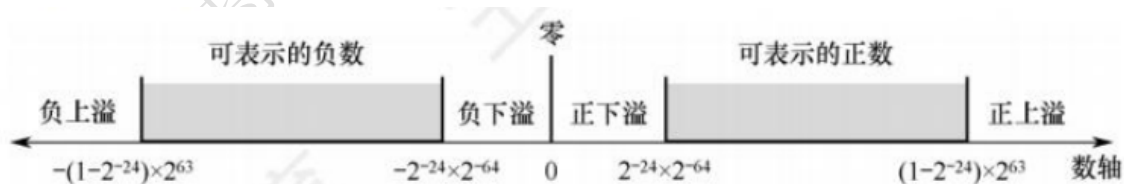
- 概念：
 - **S**：
 - 表示浮点数的符号；
 - 取值为 **0** 或 **1**；
 - **M**：
 - 表示尾数；
 - 是一个二进制的定点小数，一般用定点**原码**小数表示；
 - 尾数的位数表示了浮点数的**精度**；
 - **E**：
 - 表示阶码或**指数**；
 - 是一个二进制定点整数，用**移码**表示；
 - 其表示小数点的实际位置；
 - 阶码的范围反映了浮点数的表示**范围**；
 - **R**：
 - 表示基数，可以约定为 **2、4、16** 等；
- 解释：
 - 第 0 位：
 - 为数符 S；
 - 第 1~8 位：
 - 为 8 位移码；
 - 表示阶码 E (偏置常数为 128)；
 - SP 规格化阶码范围为 **00000001 (-126)~1111 1110 (127) -> 1~254**
 - 第 9~31 位：
 - 为 24 位二进制原码小数表示的尾数 M；
 - 规格化尾数的**小数点后第一位总是 1**，故规定第一位默认的“1”不明显表示出来，这样可用 23 个数位表示 24 位尾数；
- 图示：



浮点数的表示范围

概念：浮点数的表示范围

- 概念：
 - 浮点数的范围是关于原点对称的；
 - 浮点数的范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀；
- 机器 0：
 - 尾数为 0 或落在下溢区中的数；
- 上溢：
 - 运算结果**大于最大正数**时称为**正上溢**，小于绝对值最大负数时称为负上溢；
 - 正上溢和负上溢统称上溢；
 - 数据一旦产生上溢，计算机必须**中断**运算操作，进行溢出处理；
- 下溢：
 - 当运算结果在 0 至最小正数之间时称为正下溢；
 - 当运算结果在 0 至绝对值最小负数之间时称为负下溢；
 - 当数据下溢出时，浮点数趋于零，计算机将其当作 0 处理；
- 真题：2012年、2018年
 - 概念：浮点数的范围
 - $1.0 \times 2^{-126} \approx 1.1755 \times 10^{-38} \leq FloatNum \leq (2 - 2^{-23}) \times 2^{127} \approx 3.4028 \times 10^{38}$
 - $-(2 - 2^{-23}) \times 2^{127} \approx -3.4028 \times 10^{38} \leq FloatNum \leq -1.0 \times 2^{-126} \approx -1.1755 \times 10^{-38}$



- 解析：

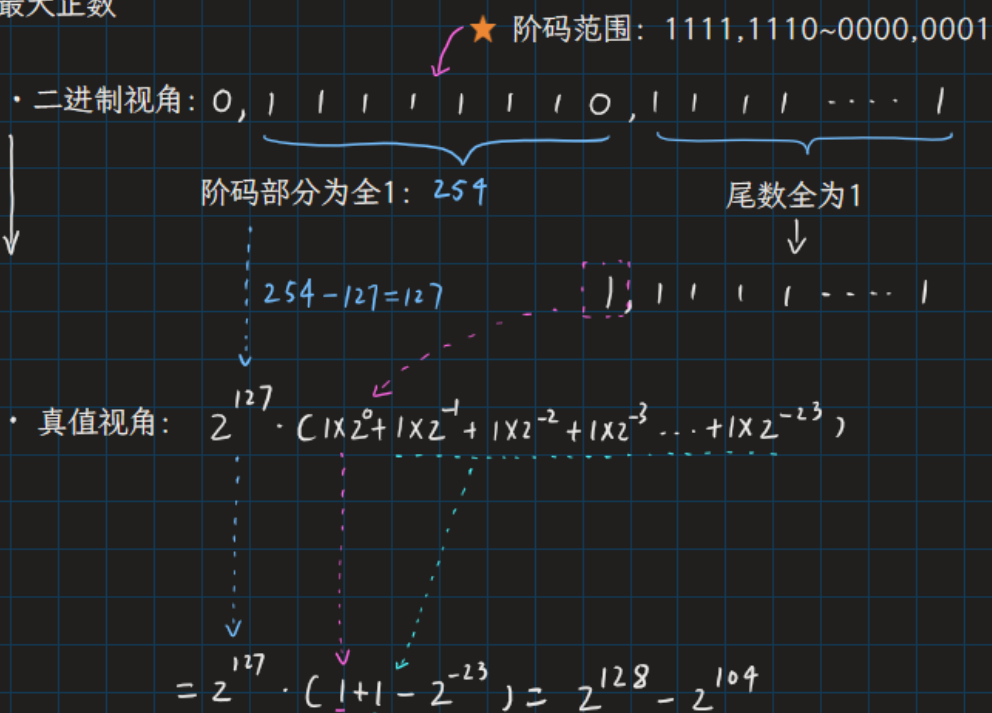
【2012 统考真题】float 型（即 IEEE 754 单精度浮点数格式）能表示的最大正整数是（ ）。

- A. $2^{126} - 2^{103}$ B. $2^{127} - 2^{104}$ C. $2^{127} - 2^{103}$ D. $2^{128} - 2^{104}$

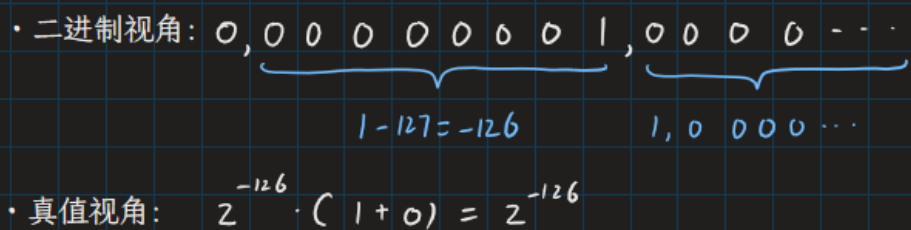
【2018 统考真题】IEEE 754 单精度浮点格式表示的数中，最小的规格化正数是（ ）。

- A. 1.0×2^{-126} B. 1.0×2^{-127} C. 1.0×2^{-128} D. 1.0×2^{-149}

(1) 最大正数



(2) 最小正数



浮点数的规格化

概念：规格化

- Normalized form (规格化形式)：小数点前只有一位非 0 数，并且规格化的数字、其形式是唯一的

概念：浮点数的规格化

- 功能：
 - 为了在浮点数运算过程中尽可能多的保留有效数字的位数，使得有效数字尽量占满尾数数位，必须在运算过程中对浮点数进行规格化操作；
- 规格化操作：
 - 通过调整一个非规格化浮点数的尾数和阶码的大小，使非零浮点数在尾数的最高数位上保证是一个有效值；

- 形式:

• $[1.XXXXX]_{\text{小数}}$

- 左规:

- 当运算结果的尾数的最高数位不是有效位, 即出现 $\pm 0.0\dots 0x\dots$ 的形式时, 需要逃行左规;
- 左规时, 尾数每左移一位、阶码减 1 (基数为 2 时);
- 左规可能要进行多次;

- 右规:

- 当运算结果的尾数的有效位进到小数点前面时, 需要进行右规, 右规只需进行一次;
- 将尾数右移一位、阶码加 1 (基数为 2 时);
- 右规时, 阶码增加可能导致溢出;

IEEE 754 标准

概念: IEEE 754 标准

- 概念:

- 二进制浮点数的尾数的最高位 (小数点第一位) 总是等于 1;
- 尾数:
 - 用原码表示;
- 指数:
 - 用移码表示, 所以在存储指数 (阶码) 前, 需要先 **将偏置值加到阶码的真值上**;
 - 偏置值为 $2^{n-1} - 1$ $\rightarrow$ 单精度: 127 ; 双精度 $\rightarrow$ 1023 ;
 - 举例: 阶码值为 3, 所以其单精度移码表示时:

$$E = 127 + X = 127 + 3 = 130 = 82H$$

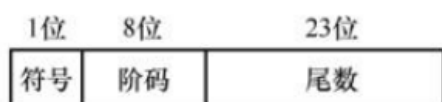
- 隐藏位:

- 为了可以使得尾数多表示一位有效位, 将这个 1 隐藏, 称之为 **隐藏位**;
- 所以 23 位尾数实际上表示了 24 位有效数字;
- 单精度与双精度浮点数都采用隐藏尾数最高位的方法, 因而使浮点数的精度更高;

- 举例:

- $[12]_{10} = [1100]_2 = f([1100]_2)_{\text{规格化}} = 1.1 * 2^3$, 且小数位的第一个 1 不存储在尾数内存中

- 图示:



(a) 32位单精度格式



(b) 64位双精度格式

类型	符号 s	阶码 e	尾数 f	总位数	偏置值	
					十六进制	十进制
单精度	1	8	23	32	7FH	127
双精度	1	11	52	64	3FFH	1023

- 注意双精度的阶码是 11 位；

规格化浮点数 -> 真值

- 单精度公式：

$$(-1)^S * 1.f * 2^{e-127}$$

- 双精度公式：

$$(-1)^S * 1.f * 2^{e-1023}$$

表示范围

- 单精度与双精度表示范围：

格式	最小值	最大值
单精度	$e = 1, f = 0$ $1.0 \times 2^{1-127} = 2^{-126}$	$e = 254, f = .111\dots, 1.111\dots1 \times 2^{254-127} = 2^{127} \times (2 - 2^{-23})$
双精度	$e = 1, f = 0$ $1.0 \times 2^{1-1023} = 2^{-1022}$	$e = 2046, f = .1111\dots, 1.111\dots1 \times 2^{2046-1023} = 2^{1023} \times (2 - 2^{-52})$

- 特殊值：

值的类型	单精度 (32 位)				双精度 (64 位)			
	符号	阶码	尾数	值	符号	阶码	尾数	值
正零	0	0	0	0	0	0	0	0
负零	1	0	0	-0	1	0	0	-0
正无穷大	0	255 (全 1)	0	∞	0	2047 (全 1)	0	∞
负无穷大	1	255 (全 1)	0	$-\infty$	1	2047 (全 1)	0	$-\infty$

值的类型	单精度 (32 位)				双精度 (64 位)			
	符号	阶码	尾数	值	符号	阶码	尾数	值
无定义数 (非数)	0 或 1	255 (全 1)	$f \neq 0$	NaN	0 或 1	2047 (全 1)	$f \neq 0$	NaN
非规格化正数	0	0	$f \neq 0$	$2^{-126}(0.f)$	0	0	$f \neq 0$	$2^{-1022}(0.f)$
非规格化负数	1	0	$f \neq 0$	$-2^{-126}(0.f)$	1	0	$f \neq 0$	$-2^{-1022}(0.f)$

- 概念：非规格化正数/负数

- 真题：2023年

- 当出现全 0 阶码、尾数非 0 时，此时是非规格化正数/负数；
- 非规格化数可以用于处理阶码下溢，使得出现比最小规格化还小的数时，程序依然能正确运行下去；

- 概念：无定义数

- 当出现全 1 阶码、尾数非 0 时，此时是无定义数，即 NaN；

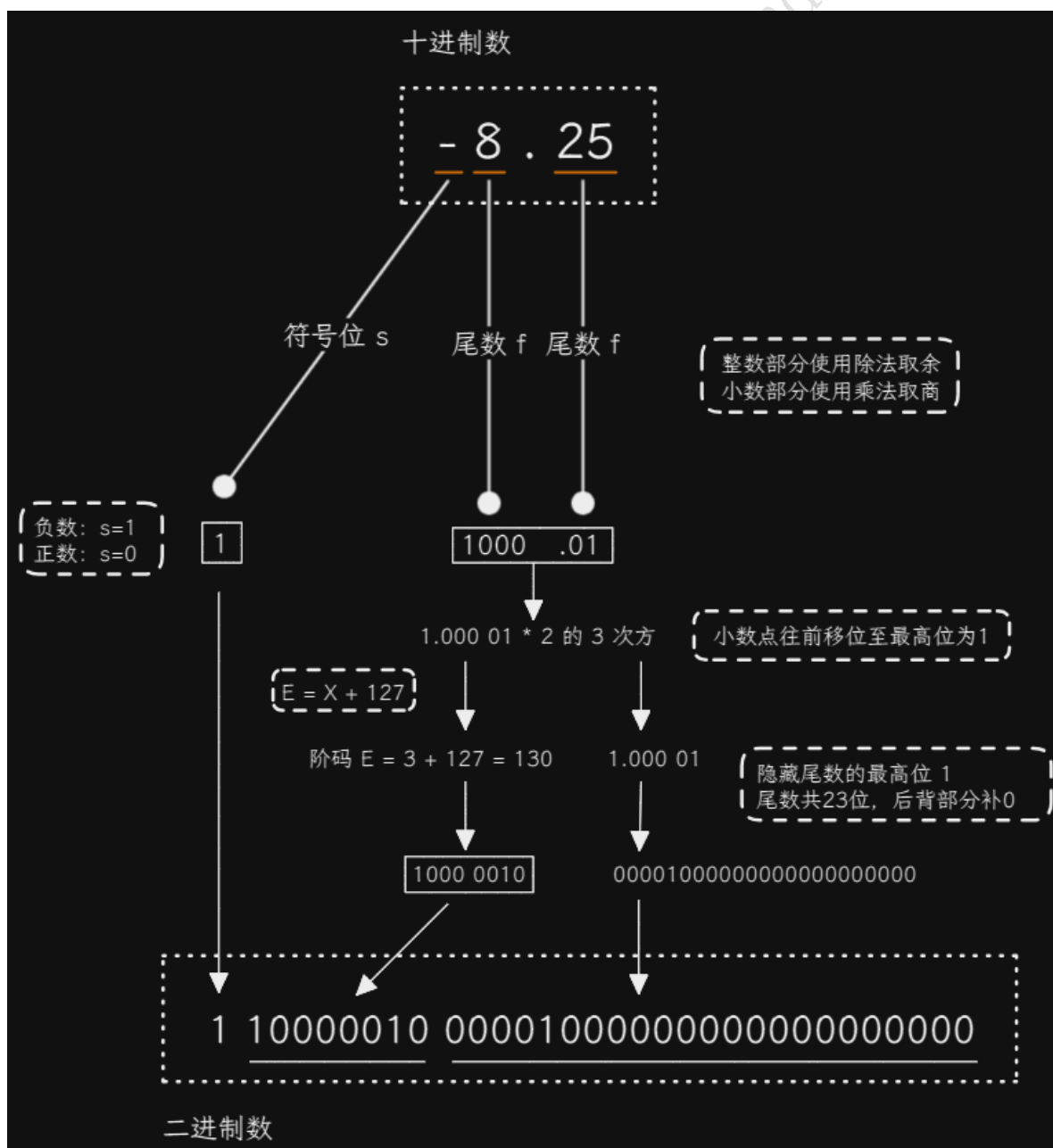
• 注意：

- 真题：2024年
- 浮点数的表示范围不代表其浮点数整型部分的范围，因为表示范围不等于精度；
- 如果想要存储一个很大的整数，其精度范围是由尾数部分尾数的位数决定的；
- 比如：想要存储一个整型，整型的范围为 2^{-40} 至 2^{40} ，则此时单精度浮点类型无法表示此数，因为单精度浮点类型的尾数部分只有 23 位，加上隐藏位就是 24 位，其精度不足以表示 40 位的整型参数。但 double 的位数部分则可以；

IEEE 单精度数与实数转换

例题：将十进制数 $[-8.25]_{(10)}$ 转化为 IEEE 754 的单精度浮点数格式

- 分析
 - 单精度浮点数的偏置值是 127；
 - 尾数最高位的“1”是被隐藏的；
- 解析



补充：二进制小数 $\leftrightarrow$ 真值

- 已知真值小数 $\rightarrow$ 二进制小数：乘法求商；
- 已知二进制小数 $\rightarrow$ 真值小数 $\rightarrow$ 反向乘法取商
 - 举例： $[0.001]_B$
 - 1： $1.0/2=0.5$
 - 0： $0.5/2=0.25$
 - 0： $0.25/2=0.125$
 - 举例： $[0.011]_B$
 - 1： $1.0/2=0.5$
 - 1： $(0.5 \rightarrow 1.5) 1.5/2=0.75$
 - 0： $0.75/2=0.375$

定点、浮点表示的区别

区别：数值表示范围

- 概念：若定点数和浮点数的字长相同，则浮点表示法所能表示的**数值范围**远大于定点表示法；

区别：精度

- 概念：对于字长相同的定点数和浮点数，浮点数虽然扩大了数的表示范围，但却**降低了精度**；

区别：数的运算

- 概念：浮点数的运算比定点数更复杂
- 原因：因为浮点数需要进行尾数、阶码的运算，并且还需要进行规格化；

区别：数的溢出

- 定点运算：
 - 当运算结果超出数的表示范围时，发生溢出；
- 浮点运算：
 - 运算结果超尾数表示范围不一定溢出；
 - 只有在规格化后，若阶码的大小超出所能表示的范围时，才发生溢出；

浮点数的加减运算

核心：浮点数的运算需要将阶码运算和尾数运算分开进行

步骤一：对阶

概念

- 目的：
 - 将两个操作数的小数点位置对其 $\rightarrow$ 使得其**阶码相等**；
- 步骤：

- 先求阶码的差，然后以**小阶码向大阶码看齐**的原则，将阶码小的尾数右移一位，阶码加 1，直到阶码相等为止；
- 尾数右移时，若舍弃有效位会产生误差，影响精度。为了保证运算的精度，尾数右移时，**低位移出的位不要丢掉**，应保留并参加尾数部分的运算
- 注意：
 - 真题：2015年
 - 在进行浮点数加减运算的对阶过程中，对阶操作不会引起阶码的溢出，并且运算时还会保留应当移出的位；

步骤二：尾数加减

概念

- 步骤：
 - 将对阶后的尾数按**定点原码小数的加（减）运算**规则进行运算；
- 注意：
 - 1. 因为 IEEE 754 浮点数尾数中有一个隐藏位，因此在进行尾数加减时，必须将**隐藏位还原到尾数**部分；
 - 2. 运算后的尾数不一定是规格化的，因此，浮点数的加减运算需要进行**规格化处理**；

步骤三：尾数规格化

概念

- IEEE 754 的规格化形式：
- $[\pm 1. \times \dots \times]_{\text{尾数}}$
- 第一位为 1

举例：右规格化

- 演示：
 - $[\pm 1 \times . \times \dots \times]_{\text{未规格化尾数}} \rightarrow [\pm 1. \times \times \dots \times]_{\text{规格化尾数}}$
- 分析：
 - 小数点前面有两位，将尾数右移一位，并且阶码加 1
 - 最高位 1 被移动到小数点的前一位，作为隐藏位；
 - 同理左规格化；
- 注意：
 - 1. 左规一次相当于乘以 2，右规一次相当于除以 2；
 - 2. 左移可能需要多次，需一直左移至第一位 1 在小数点左边。右移只需要一次；
 - 真题：2015年
 - 3. 右规格化可能会引起阶码上溢；

步骤四：尾数舍入

概念

- 原因：
 - 在对阶和尾数右规时，可能会对尾数进行**右移**；
 - 为保证运算精度，一般将移出的部分低位保留下来，参加中间过程的运算，**最后再将运算结果进行舍入**，还原表示成 `IEEE 754` 格式；
- 舍入模式：
 - (1) 就近舍入：
 - 舍入为最近的可表示数；
 - 概念：
 - 当运算结果是两个可表示数的非中间值时，实际上是“0 舍 1 入”方式
 - 类似于十进制的“四舍五入”法；
 - 当运算结果正好在**两个可表示数的中间**时，则选择结果为**偶数**；
 - 举例：
 - 计算 $1.24 \times 10^5 + 5.04 \times 10^2$ ；
 - 若只保留 2 位保存位；
 - $1.2400 \times 10^5 + 0.0050 \times 10^5 = 1.2450 \times 10^5$
 - 因为结果 1.2450×10^5 在 1.24×10^5 、 1.25×10^5 两数之间，因为使用舍入到偶数，所以结果为：
 - 1.24×10^5
 - (2) 正向舍入：
 - 朝数轴 $+\infty$ 方向舍入；
 - 即取右边最近的可表示数；
 - (3) 负向舍入：
 - 朝数轴 $-\infty$ 方向舍入；
 - 即取左边最近的可表示数；
 - (4) 截断法：
 - 直接截取所需位数，丢弃后面的所有位；
 - 这种舍入处理最简单；
 - 对正数或负数来说，都是取更接近原点的那个可表示数，是一种趋向原点的舍入；
- 注意：
 - 尾数溢出时，结果不一定溢出，因为可以选择尾数舍入、通过减少精度来防止溢出，也可以通过调整阶码、来防止溢出；

总结：溢出判断

概念：溢出判断

- 原因：
 - 在尾数规格化和尾数舍入时，可能会对**结果的阶码**执行加减运算；
 - 比如在右规格化的时候，需要相应的增加其阶码，因此可能会超出阶码的表示范围；

- 因此必须考虑指数溢出问题;
- 概念: 两种溢出
 - 阶码溢出:
 - 若一个正指数超过了最大允许值 (127 或 1023), 则发生**指数上溢**, 产生**异常**;
 - 若一个负指数超过了最小允许值 (-149 或 -1074), 则发生指数下溢, 通常把结果**按机器零**处理;
 - 尾数溢出:
 - 1. 右规和尾数舍入:
 - 数值很大的尾数舍入时, 可能因为末位加 1 而发生尾数溢出, 此时需要通过右规来调整尾数和阶码;
 - 右规时阶码加 1, 导致阶码增大, 因此需要判断是否发生了**指数上溢**;
 - 当调整前的阶码为 11111110 时, 加 1 后会变成 11111111 而发生指数上溢;
 - 2. 左规:
 - 左规时阶码减 1, 导致阶码减小, 因此需要判断是否发生了**指数下溢**;
 - 其判断规则与指数上溢类似, 左规一次, 阶码减 1, 然后判断阶码是否为全 0 来确定指数是否下溢;
- 总结:
 - 1. 浮点数的溢出并不是以尾数溢出来判断的;
 - 2. **尾数溢出**可以通过右规操作得到纠正;
 - 3. **运算结果溢出**主要看结果的指数是否发生了上溢, 因此是由指数上溢来判断的;

C 语言中的浮点数类型

数据类型转换

概念

- 类型:
 - float -> IEEE 754 单精度浮点类型;
 - double -> IEEE 754 双精度浮点类型;
 - long double -> IEEE 754 扩展双精度浮点类型;
- 注意:
 - long double 长度和格式随编译器和处理器类型的不同;
- 无损失转换:
 - 1. char -> int -> long -> double
 - 2. float -> double

混合运算: 隐式类型转换

- 概念:
 - 不同类型进行混合运算时, 将**较低类型转化为较高类型**;
- 举例:

- 隐式类型转换: `int` 和 `long` 运算时, `int->long`

类型精度范围分析

分析: `int->float`

- `float` 的尾数时 24 位;
- 当 `int` 数的第 24~31 位非 0 时, 无法精确转化为 24 位浮点数尾数 -> 舍入;

分析: `int->double`

- `double` 总能充分容纳 `int` 型的数据;

分析: `double->float`

- 在进行大数转换时, 可能发生溢出;

分析: `float/double->int`

- 因 `int` 型没有小数部分, 因此数据会向 0 方向截断 (仅保留整数部分), 发生舍入;
- 另外, 因 `int` 型的表示范围更小, 因此大数转换时可能会溢出;

获取笔记持续更新可加微信: VvaddD2277