# 整数

#### 一个整数。

该数字可以是负数、零或正数。由于 Typst 使用 64 位来存储整数,因此整数不能小于或大于。-92233720368547758089223372036854775807

该数字还可以指定为十六进制、八进制或二进制,方法是以零开头,后跟x、o或b。 您可以使用该类型的构造函数将值转换为整数。

# 例子

```
#(1 + 2) \
#(2 - 5) \
#(3 + 4 < 8)

#0xff \
#0o10 \
#0b1001

3
-3
true
255
8
9
```

# 构造函数 ❷

将值转换为整数。

- 布尔值转换为0or 1。
- 浮点数取整至下一个64位整数。

value 布尔值 或者 整数 或者 漂浮 或者 斯特 必需的 位置性 €

- 字符串以 10 为基数进行解析。

```
整数 ( 布尔值 整数 漂浮 斯特 ) -> 整数

#int(false) \
#int(true) \
#int(2.7) \
#(int("27") + int("4"))

2
31
```

应转换为整数的值。

### 定义 @

### signum

计算整数的符号。

- 如果数字为正数,则返回1。
- 如果数字为负数,则返回。-1
- 如果数字为零,则返回⊙。

自己。符号()->整数

```
#(5).signum() \
#(-5).signum() \
#(0).signum() \
```

1 -1 0

#### bit-not

计算整数的按位 NOT。

就该函数而言,操作数被视为64位有符号整数。

自己。位非()->整数

#4.bit-not() #(-1).bit-not()

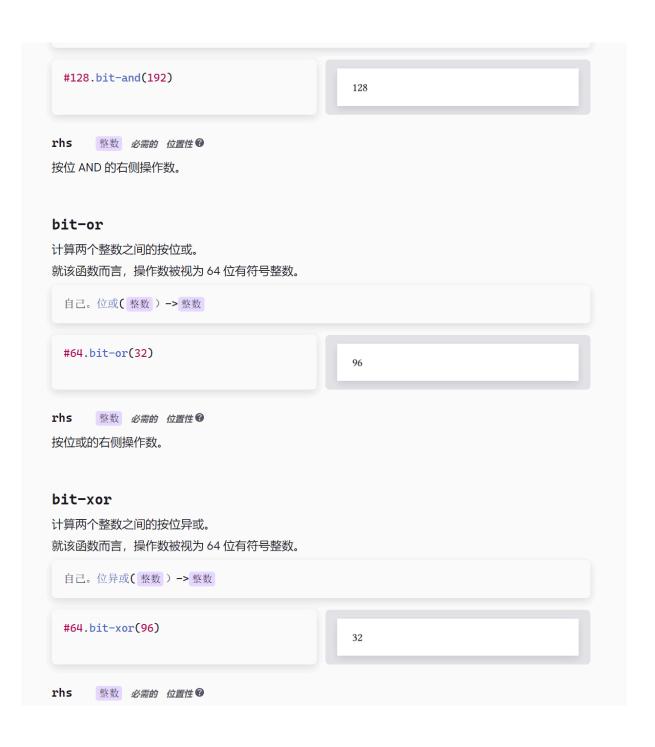
-50

#### bit-and

计算两个整数之间的按位与。

就该函数而言,操作数被视为 64 位有符号整数。

自己。位与(整数)->整数



按位异或的右侧操作数。

#### bit-lshift

将操作数的位向左移动指定的量。

就该函数而言,操作数被视为 64 位有符号整数。如果结果太大而无法容纳 64 位整数,则会出现错误。

自己。位左移(整数)->整数

#33.bit-lshift(2)
#(-1).bit-lshift(3)

132 - 8

shift 整数 必需的 位置性 ②

要移位的位数。一定不能为负数。

#### bit-rshift

将操作数的位向右移动指定的量。默认情况下执行算术移位(将符号位向左扩展,使负数保持负数),但可以通过参数更改logical。

就该函数而言,操作数被视为64位有符号整数。

自己。位右移(

整数

逻辑: 布尔值,

) -> 整数

#64.bit-rshift(2)

#(-8).bit-rshift(2)

#(-8).bit-rshift(2, logical: true)

16 -2 4611686018427387902

shift 整数 必需的 位置性®

要移位的位数。一定不能为负数。

允许大于 63 的移位,这将导致返回值饱和。对于非负数,返回值在处饱和0,而对于负数,-1如果logical设置为false,则0返回值在处饱和true。此行为与多次应用此操作一致。因此,转变总会成

功。

### logical 布尔值

切换是否应执行逻辑(无符号)右移而不是算术右移。如果是true,则负操作数将不会保留其符号位,并且移位后出现在左侧的位将为0。该参数对非负操作数没有影响。

默认: false

**く** 功能

标签 >