# int类型的数据，能玩哪些运算

int作为C/C++中最基础的内置整数类型，是数值运算的核心载体。从简单的加减乘除，到复杂的位运算、复合赋值运算，int类型的运算覆盖日常开发、算法实现、底层编程等多类场景。看似基础的运算操作，不仅有语法规范需遵循，更暗藏溢出、优先级、符号适配等诸多细节。本文将立足int类型，全面拆解各类运算的用法、场景与避坑技巧，结合代码示例帮你吃透int运算的全场景玩法，衔接此前IO格式控制内容，形成完整的int类型使用体系。

## 一、基础运算：算术运算，数值计算的基石

算术运算是int类型最核心、最常用的运算，用于实现整数间的加减乘除、取余，适配基础数值计算场景，语法简洁但需注意溢出与取余规则。

### 1. 核心算术运算符

int类型支持5类基础算术运算符，分别对应加法、减法、乘法、除法、取余，运算结果默认仍为int类型，需注意运算规则与数据范围限制。

| 运算符 | 作用 | 示例 | 说明 |
| --- | --- | --- | --- |
| + | 加法 | a + b | 两个int变量求和，支持正数、负数、零的混合运算 |
| - | 减法 | a - b | 求两个int变量的差值，可用于数值递减 |
| * | 乘法 | a * b | 两个int变量求积，易触发溢出，需关注数据范围 |
| / | 除法 | a / b | 整数除法，舍弃小数部分（非四舍五入），除数不能为0 |
| % | 取余（模运算） | a % b | 求除法余数，结果符号与被除数一致，除数不能为0 |

### 2. 代码示例：基础算术运算实操

```
#include <iostream>
using namespace std;

```

```
int main() {
    int a = 15, b = 4;

    // 加法
    cout << a << " + " << b << " = " << a + b << endl; // 输出：15 + 4 = 19

    // 减法
    cout << a << " - " << b << " = " << a - b << endl; // 输出：15 - 4 = 11

    // 乘法
    cout << a << " * " << b << " = " << a * b << endl; // 输出：15 * 4 = 60

    // 除法（舍弃小数）
    cout << a << " / " << b << " = " << a / b << endl; // 输出：15 / 4 = 3（非3.75）

    // 取余（余数符号与被除数一致）
    cout << a << " % " << b << " = " << a % b << endl; // 输出：15 % 4 = 3
    cout << (-a) << " % " << b << " = " << (-a) % b << endl; // 输出：-15 % 4 = -3

    return 0;
}
```

## 3. 关键避坑点

- **除法避坑**：int类型除法仅保留整数部分，无论小数部分是否大于0.5（如15/4=3、17/4=4）；若需保留小数，需将其中一个操作数转为float或double类型。
- **取余避坑**：除数不能为0，否则会触发运行时错误；余数符号与被除数一致，而非除数（如-15%4=-3、15%-4=3）。
- **溢出避坑**：int类型有固定取值范围（通常为$-2^{15}$~$2^{15}-1$，即-32768~32767；32位系统中为$-2^{31}$~$2^{31}-1$），运算结果超出范围会触发溢出（未定义行为），需提前判断或使用long long类型。

```
#include <iostream>
using namespace std;

int main() {
    // 溢出示例：int最大值+1，结果异常
    int max_int = INT_MAX; // INT_MAX是int类型最大值，需包含<climits>头文件
    cout << "int最大值：" << max_int << endl;
    cout << "int最大值+1：" << max_int + 1 << endl; // 输出最小值，触发溢出

```

```
    // 除法保留小数的正确写法
    int a = 15, b = 4;
    cout << "保留小数的除法结果：" << (double)a / b << endl; // 强制转换后输出3.75

    return 0;
}

```

## 二、进阶运算：复合赋值运算，简化代码的利器

复合赋值运算是算术运算与赋值运算的结合，核心作用是简化代码（如将a = a + b简化为a += b），同时提升代码可读性，是日常开发中高频使用的运算方式。

### 1. 常用复合赋值运算符

int类型支持6类复合赋值运算符，均是“算术运算+赋值”的缩写，运算逻辑与单独运算一致，仅语法更简洁。

| 复合运算符 | 等价写法 | 说明 |
| --- | --- | --- |
| += | a = a + b | 将a与b的和赋值给a |
| -= | a = a - b | 将a与b的差赋值给a |
| * | a = a * b | 将a与b的积赋值给a |
| /= | a = a / b | 将a与b的商赋值给a，遵循整数除法规则 |
| %= | a = a % b | 将a与b的余数赋值给a，遵循取余规则 |

### 2. 代码示例：复合赋值运算实操

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;

    // += 运算
    a += 5; // 等价于a = a + 5
    cout << "a += 5 后：" << a << endl; // 输出：15

```

```
    // -= 运算
    a -= 3; // 等价于a = a - 3
    cout << "a -= 3 后: " << a << endl; // 输出: 12

    // *= 运算
    a *= 2; // 等价于a = a * 2
    cout << "a *= 2 后: " << a << endl; // 输出: 24

    // /= 运算
    a /= 4; // 等价于a = a / 4
    cout << "a /= 4 后: " << a << endl; // 输出: 6

    // %= 运算
    a %= 4; // 等价于a = a % 4
    cout << "a %= 4 后: " << a << endl; // 输出: 2

    return 0;
}

```

### 3. 注意事项

复合赋值运算符的优先级低于算术运算符，若右侧是复杂表达式，建议添加括号明确优先级，避免运算逻辑错误。

```
#include <iostream>
using namespace std;

int main() {
    int a = 5;
    // 注意优先级: a += 3 * 2 等价于 a = a + (3*2)，而非 (a+3)*2
    a += 3 * 2;
    cout << "a += 3*2 后: " << a << endl; // 输出: 11

    // 若需实现(a+3)*2，需添加括号
    a = 5;
    a = (a + 3) * 2;
    cout << "(a+3)*2 后: " << a << endl; // 输出: 16

    return 0;
}
```

## 三、进阶运算：自增自减运算，快速增减的实用技巧

自增（++）、自减（--）是int类型特有的运算，用于实现变量自身加1或减1，分为前置和后置两种形式，适配循环计数、数值迭代等场景，用法灵活但需区分前置与后置的差异。

### 1. 自增自减的两种形式

- **前置形式（++a、--a）**：先对变量执行加1/减1运算，再使用变量的值（先运算，后取值）。
- **后置形式（a++、a--）**：先使用变量当前的值，再对变量执行加1/减1运算（先取值，后运算）。

### 2. 代码示例：前置与后置的差异

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 5;

    // 前置自增：先加1，后取值
    cout << "前置自增 ++a：" << ++a << endl; // 输出：6，此时a=6
    cout << "前置自增后a的值：" << a << endl; // 输出：6

    // 后置自增：先取值，后加1
    cout << "后置自增 b++：" << b++ << endl; // 输出：5，此时b仍为5
    cout << "后置自增后b的值：" << b << endl; // 输出：6

    // 自减运算同理
    int c = 5, d = 5;
    cout << "前置自减 --c：" << --c << endl; // 输出：4，c=4
    cout << "后置自减 d--：" << d-- << endl; // 输出：5，后续d=4

    return 0;
}

```

### 3. 适用场景与避坑

- **循环场景**：for循环中，前置与后置自增效果一致（如for(int i=0; i<10; i++)与for(int i=0; i<10; ++i)），均可实现计数。
- **赋值场景**：需注意取值与运算的顺序，避免逻辑错误（如int x = a++，x得到a的原值，而int x = ++a，x得到a加1后的值）。
- **效率提示**：对于int类型，前置与后置自增效率差异极小；但对于自定义类型（如结构体），前置自增效率更高（无需创建临时变量）。

## 四、高阶运算：位运算，底层开发的核心技能

位运算是直接对int变量的二进制位进行操作的运算，效率极高，适配底层开发、算法优化、权限控制等场景（如二进制掩码、状态标记）。int类型的位运算基于其二进制存储形式（补码），需熟悉二进制与十进制的转换规则。

## 1. 常用位运算符

| 运算符 | 作用 | 示例（a=6(0110)，b=3(0011)） | 结果（二进制） | 结果（十进制） |
|---|---|---|---|---|
| & | 按位与 | a & b | 0010 | 2 |
| \| | 按位或 | a \| b | 0111 | 7 |
| ^ | 按位异或 | a ^ b | 0101 | 5 |
| ~ | 按位非 | ~a | 1001（补码） | -7 |
| << | 左移 | a << 1 | 1100 | 12 |
| >> | 右移 | a >> 1 | 0011 | 3 |

## 2. 核心位运算用法解析

```
#include <iostream>
using namespace std;

int main() {
    int a = 6; // 二进制：0000 0110（简化为8位）
    int b = 3; // 二进制：0000 0011

    // 1. 按位与：对应位都为1，结果为1（常用于清0特定位）
    cout << "a & b = " << (a & b) << endl; // 输出：2（0000 0010）

    // 2. 按位或：对应位有一个为1，结果为1（常用于置1特定位）
    cout << "a | b = " << (a | b) << endl; // 输出：7（0000 0111）

    // 3. 按位异或：对应位不同为1，相同为0（常用于交换变量、翻转位）
    cout << "a ^ b = " << (a ^ b) << endl; // 输出：5（0000 0101）
    // 异或交换变量（无需临时变量）
    int x = 10, y = 20;
    x ^= y;
    y ^= x;
    x ^= y;
    cout << "交换后 x=" << x << ", y=" << y << endl; // 输出：x=20, y=10

```

```
    // 4. 按位非：按位取反（0变1，1变0），结果为原数+1的负数
    cout << "~a = " << ~a << endl; // 输出：-7（补码机制）

    // 5. 左移：整体左移n位，右侧补0，等价于a*2^n（效率高于乘法）
    cout << "a << 1 = " << (a << 1) << endl; // 输出：12（6*2^1）

    // 6. 右移：整体右移n位，左侧补符号位，等价于a/2^n（整数除法）
    cout << "a >> 1 = " << (a >> 1) << endl; // 输出：3（6/2^1）

    return 0;
}

```

### 3. 位运算避坑要点

- **符号位问题**：int是带符号类型，右移时左侧补符号位（正数补0，负数补1），可能导致结果异常；无符号int（unsigned int）右移时左侧补0。
- **左移溢出**：左移可能导致二进制位超出int范围，触发溢出（未定义行为），需提前判断数据范围。
- **按位非规则**：按位非运算结果遵循补码机制，并非简单的“原数取反”，如~6 = -7（而非9）。

## 五、逻辑运算：判断与分支的核心支撑

int类型可参与逻辑运算，核心用于条件判断（如if、while语句），逻辑运算的结果是布尔值（true=1，false=0）。需注意：非0的int值均被视为true，0被视为false。

### 1. 常用逻辑运算符

| 运算符 | 作用 | 示例 | 说明 |
| --- | --- | --- | --- |
| & | 逻辑与 | a && b | a和b均为非0（true），结果为1，否则为0（短路求值） |
| \|\| | 逻辑或 | a \|\| b | a和b至少一个为非0（true），结果为1，否则为0（短路求值） |
| ! | 逻辑非 | !a | a为0（false），结果为1；a为非0（true），结果为0 |

### 2. 代码示例：逻辑运算实操

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 0, c = -3;

    // 逻辑与：短路求值，左侧为false则不执行右侧
    cout << "a && b = " << (a && b) << endl; // 输出：0（b为0，false）
    cout << "a && c = " << (a && c) << endl; // 输出：1（均为非0，true）

    // 逻辑或：短路求值，左侧为true则不执行右侧
    cout << "a || b = " << (a || b) << endl; // 输出：1（a为非0，true）
    cout << "b || 0 = " << (b || 0) << endl; // 输出：0（均为false）

    // 逻辑非：取反布尔值
    cout << "!a = " << !a << endl; // 输出：0（a非0，取反为false）
    cout << "!b = " << !b << endl; // 输出：1（b为0，取反为true）

    // 逻辑运算用于条件判断
    if (a && c) {
        cout << "a和c均为非0" << endl;
    }
    if (!b) {
        cout << "b为0" << endl;
    }

    return 0;
}

```

### 3. 关键注意点

- **短路求值**：逻辑与（&&）左侧为false时，右侧表达式不执行；逻辑或（||）左侧为true时，右侧表达式不执行，需避免在右侧写修改变量的操作（如a++）。
- **与位运算区分**：逻辑与（&&）、逻辑或（||）与按位与（&）、按位或（|）完全不同，前者返回布尔值，后者返回整数，不可混淆。

## 六、比较运算：判断大小的基础操作

比较运算用于判断两个int变量的大小关系，运算结果为布尔值（true=1，false=0），是条件判断、循环控制的核心，与逻辑运算结合可实现复杂判断逻辑。

### 1. 常用比较运算符

| | | | |
|---|---|---|---|

| 运算符 | 作用 | 示例 | 结果（a=5，b=3） |
| --- | --- | --- | --- |
| > | 大于 | a > b | 1（true） |
| < | 小于 | a < b | 0（false） |
| >= | 大于等于 | a >= b | 1（true） |
| <= | 小于等于 | a <= b | 0（false） |
| == | 等于 | a == b | 0（false） |
| != | 不等于 | a != b | 1（true） |

## 2. 代码示例：比较运算实操

```
#include <iostream>
using namespace std;

int main() {
    int a = 10, b = 15;

    cout << a << " > " << b << " : " << (a > b) << endl; // 0
    cout << a << " < " << b << " : " << (a < b) << endl; // 1
    cout << a << " == " << b << " : " << (a == b) << endl; // 0
    cout << a << " != " << b << " : " << (a != b) << endl; // 1
    cout << a << " >= " << 10 << " : " << (a >= 10) << endl; // 1

    // 比较运算结合逻辑运算，实现复杂判断
    int score = 85;
    if (score >= 60 && score < 90) {
        cout << "成绩为良好" << endl;
    }

    return 0;
}

```

## 3. 避坑要点

- **避免混淆==与=**：==是比较运算符（判断是否相等），=是赋值运算符（给变量赋值），写错会导致逻辑错误（如if(a=5)会给a赋值5，永远为true）。
- **溢出影响比较**：两个int变量运算后比较时，需避免运算溢出导致比较结果异常（如INT_MAX + 1 < 0，会误判为true）。

## 七、总结：int运算的全场景用法与核心原则

int类型的运算覆盖“基础计算-代码简化-底层操作-条件判断”四大维度，各类运算各有适配场景，核心用法可归纳为三类：

1. 基础运算（算术、比较）：满足日常数值计算、大小判断，是所有运算的基础，需重点规避溢出、除法取整、比较符号混淆等问题；
2. 简化运算（复合赋值、自增自减）：精简代码，提升可读性，适配循环迭代、数值更新场景，需区分自增自减的前置与后置差异；
3. 高阶运算（位运算、逻辑运算）：位运算适配底层开发、算法优化，逻辑运算适配条件判断，需熟悉二进制规则与短路求值特性。

掌握int类型的各类运算，不仅能高效完成数值处理与逻辑判断，更能规避常见错误，提升代码效率与健壮性。实际开发中，需结合场景选择合适的运算方式，同时关注数据范围，避免溢出等未定义行为，搭配此前掌握的IO格式控制技巧，可形成完整的int类型使用能力，为复杂项目开发打下坚实基础。