C++ 高精度加法

一：为什么需要高精度计算?

在编程中，我们常用的 `int`、`long long` 等数据类型都有其表示范围的上限。例如，一个 64 位的 `long long` 最大也只能存储约 `9 x 10^18` 的数字。当我们面临的问题需要处理的数值远超这个范围时——比如计算 100 的阶乘（一个有 158 位的天文数字），或者在密码学中处理超长大整数——标准数据类型便无能为力了。

高精度计算，就是为了解决这个问题而生。它利用字符串或数组来模拟我们人类进行笔算的过程，从而处理任意大（只要内存允许）的整数运算。而接下来就会讲解最基础也最重要的运算：**高精度加法。**

二：算法思路

1. 方法一：双指针法

最自然的思路是**模拟小学的竖式计算**：末位对齐，从右到左逐位相加，并处理进位。如果用代码直接实现，通常是这样的：

- 使用两个指针 `i` 和 `j`，分别指向两个字符串 `a` 和 `b` 的末尾。
- 在一个 `while (i >= 0 || j >= 0)` 的循环中进行计算。
- 循环内部需要写复杂的 `if-else` 来处理：`i` 和 `j` 都有效、只有 `i` 有效、只有 `j` 有效这三种情况。

这种方法虽然直观，但**代码实现相对复杂，边界条件多，容易出错。**

2. 反转对齐法

为了规避双指针的复杂性，我们引入一种实现更简洁的思路：

“与其费力地从后往前处理复杂的边界，不如我们先把它变成一个简单的、从头走到尾的问题！”

这就是“反转对齐法”的核心思想。它通过预处理，将所有情况（如两个数长度不同）统一为一种标准模型，使得最核心的“逐位相加”过程可以用一个极其简单的单循环完成。

三：核心流程

让我们通过计算 `987 + 56` 来感受这个这个过程：

原始数字

- `a = "987"`
- `b = "56"`

第一步：全部反转
我们将两个字符串都进行反转，这样它们的个位、十位、百位...就分别对齐到了字符串的索引 `0, 1, 2...`。

- `a` 变为 `"789"`
- `b` 变为 `"65"`

第二步：补零对齐
为了让两个字符串的每一位都能一一对应相加，我们将较短的字符串在末尾补 `0`，直到它们等长。

- `a` 保持 `"789"`

- `b` 变为 `"650"`

> 在反转后的字符串**末尾**补 `0`，等价于在原始数字的**开头**补 `0`（例如，`56` 变成 `056`），这是在模拟竖式计算的末位对齐。

第三步：从左到右逐位相加
由于已经反转对齐，我们现在可以从索引 `0` 开始，用一个简单的 `for` 循环来完成所有位的相加。

```
      (carry)   0 -> 1 -> 1 -> 1   (进位变化过程)
                7    8    9       (反转后的 a)
           +    6    5    0       (反转后的 b)
           -----------------
      (ans)     3 -> 4 -> 0       (倒序的结果 ans)
```

- **i = 0 (个位):** `carry` 是 `0`。`7 + 6 + 0 = 13`。结果存 `3`，新 `carry` 为 `1`。`ans = "3"`。
- **i = 1 (十位):** `carry` 是 `1`。`8 + 5 + 1 = 14`。结果存 `4`，新 `carry` 为 `1`。`ans = "34"`。
- **i = 2 (百位):** `carry` 是 `1`。`9 + 0 + 1 = 10`。结果存 `0`，新 `carry` 为 `1`。`ans = "340"`。

第四步：处理最终进位
循环结束后，检查是否还有剩余的进位。在这里，`carry` 等于 `1`，需要追加到结果的末尾。

- `ans` 变为 `"3401"`。

第五步：再次反转
我们得到的 `ans` 是最终结果的倒序。现在，我们把它反转回来，得到正确的答案。

- `"3401"` 反转后得到 `"1043"`。

至此，计算完成！这个过程将一个复杂的、需要处理多种情况的双指针问题，转化成了一个简单的、逻辑统一的单循环问题。

四：代码实现

```
string add(string a, string b) {
    // 将两个输入字符串都反转
    // 目的：让个位在 [0] 位置，方便从左到右处理
    reverse(a.begin(), a.end());
    reverse(b.begin(), a.end());

    // 在较短的字符串末尾补 0，使其与较长的等长
    // 目的：统一循环次数，避免复杂的边界判断
    while (a.size() < b.size()) a += '0';
    while (b.size() < a.size()) b += '0';

    string ans = "";    // 存放倒序的结果
    int carry = 0;      // 初始化进位

    // 从左到右逐位相加（i=0 是个位，i=1 是十位...）
    for (int i = 0; i < a.size(); i++) {
        // 计算当前第 i 位的总和
        int sum = (a[i] - '0') + (b[i] - '0') + carry;

        // 将结果的个位数（字符形式）拼接到 ans 的末尾
```

```
        ans += to_string(sum % 10);

        // 更新进位，为下一轮做准备
        carry = sum / 10;
    }

    // 如果循环结束后仍有进位（如 99+1=100），则追加到 ans 末尾
    if (carry > 0) {
        ans += to_string(carry);
    }

    // 将倒序的结果 ans 再次反转，得到最终答案
    reverse(ans.begin(), ans.end());

    return ans;
}
```

第五部分：总结与要点

- **核心思路**：通过 `反转 -> 补零 -> 相加 -> 再反转` 的流程，将问题简化。
- **设计思想**：**将复杂性从核心循环中剥离，移入预处理和后处理阶段。**
- 通过预处理将所有输入（无论长度如何）都统一为相同长度的字符串，避免了在核心循环中进行繁琐的长度判断，大大减少了出错的可能性。
- **技巧**：在**反转后**的字符串**末尾**补零，是实现**原始数字**在**开头**对齐的关键。