Barrett 模乘算法：C/C++ 中整数除法的高效实现

鲍辰睿

摘要

整数除法是编程语言中最常用的基础运算之一，其性能对程序运行效率有重要影响。C 和 C++ 语言应用 Barrett 模乘算法，给出了该运算的优秀实现方案。本文将介绍这种优化方法，并展示其优化效果。

1 引言

整数除法是一种特殊的除法运算，它对两个整数作商，取商的整数部分作为运算结果。例如 `4/2==2`，`5/3==1`。

整数除法是编程语言常用的基础运算之一。在常用运算中，整数除法的性能明显较差。为了直观展示这一点，下面给出相关的性能测试数据。

测试所用的程序如下:

```
#include <cstdio>
#include <ctime>

int main() {
    int n=1e8,s=1;
    int t0=clock();
    for (int i=1; i<=n; ++i) /* A */
    int t1=clock();
    printf("%d\n",s);
    printf("%.0lf ms\n",1000.*(t1-t0)/CLOCKS_PER_SEC);
    return 0;
}
```

在 A 处填入执行某种运算的语句, 则程序会执行 10^8 次该运算，并测出总用时. 测试结果如下表所示（测试平台：https://duck.ac）：

编号	运算名称	A 处语句	耗时
0	异或	`s=s^i;`	28 ms
1	加	`s=s+i;`	28 ms
2	减	`s=s-i;`	28 ms
3	乘	`s=s*i;`	83 ms
4	整除	`s=s/i;`	678 ms
5	取模	`s=s%i;`	704 ms
6	除、取模	`s=s/i+s%i;`	787 ms

表 1: 几种常用运算执行 10^8 次的平均耗时

可见，在以上整数运算中，加法、减法、位运算的运行速度最快；乘法略慢；整除和取模最慢，耗时是加减法的 $20 \sim 30$ 倍。

值得注意的是，取模慢的原因和除法基本一致，因为二者共用同一条汇编指令 (`idiv`)。若要实现 x 对 `y` 取模，该指令会先计算 `x/y`，再利用数学性质 `(x%y)==x-(x/y)*y` 得到取模结果。可见取模的性能瓶颈也在于整除。这正是表中第 4,5,6 组的耗时相差不大的原因。

总之，在各种常用运算中，整除（以及取模）的运行速度尤其慢，在大量使用时可能严重影响程序的运行效率。

下面介绍一种兼具泛用性和高效性的整除优化方式：Barrett 模乘算法。

2 Barrett 模乘算法：原理介绍

该算法适用于除数 y 固定，需要对多个被除数 x 计算整除（取模）运算的情形。

本文将省略关于负数除法的不必要细节，因此不妨设 x, y 都是正整数。

为了更自然地展现 Barrett 模乘算法的思想和原理，下面给出三种优化版本。其中第一种为简易版，而第二、三种则是标准的 Barrett 模乘算法。

2.1 整除优化：版本一

流程：预处理 $I=\dfrac{1}{y}$，则有 $\dfrac{x}{y}=xI$。要计算 `x/y`，只需计算一次浮点数乘法 `x*I`，再取整数部分即可。

原理：这一版优化相当简单，只是将除法化为乘法处理。

评价：此优化似乎过于简单而粗糙，但确实有效，因为一般而言浮点数乘法的速度比整数除法快。然而其效果还不够令人满意；更大的问题是，浮点数运算可能产生精度误差，这使结果的正确性难以得到保证。

2.2 整除优化：版本二 [1]

流程：设 $k=\lfloor\log_2 x(y-1)\rfloor+1$，预处理 $I=\left\lceil\dfrac{2^k}{y}\right\rceil$（$\lceil x\rceil$ 表示对 x 上取整；而下文中 $\lfloor x\rfloor$ 表示对 x 下取整）。要计算 `x/y` 的值，只需计算 `x*I>>k`。

原理：这其实是将版本一的 I 扩大为原来的 2^k 倍，再近似为整数存储，从而避免了浮点数运算。下面尝试分析这一近似产生的误差：

$$0\leqslant I-\frac{2^k}{y}\leqslant\frac{y-1}{y}$$

上述不等式可由 $I=\left\lceil\dfrac{2^k}{y}\right\rceil$ 以及上取整的性质得到。将其中每项乘 $\dfrac{x}{2^k}$：

$$0\leqslant\frac{xI}{2^k}-\frac{x}{y}\leqslant\frac{x(y-1)}{2^k y}$$

当 $2^k>x(y-1)$ 时，可保证误差小于 $\dfrac{1}{y}$，此时 $\left\lfloor\dfrac{xI}{2^k}\right\rfloor$ 一定等于 $\left\lfloor\dfrac{x}{y}\right\rfloor$。因此我们选取 $k=\lfloor\log_2 x(y-1)\rfloor+1$，从而保证整除运算结果的误差为 0。

评价：此优化的思想仍是化除为乘，但完全使用整数运算，避免了精度误差，正确性得到了严谨证明。并且其代码实现也十分简单。

此外，由于 `(x%y)==x-(x/y)*y`，此算法也可以用于优化取模运算。优化之后，整除、取模分别只需执行 1 次、2 次乘法。

然而，此优化也存在一些不足。以 32 位无符号整数（下简称为 `u32`）的除法、取模为例：在极限数据下，I 可能超过 `u32` 的范围，那么 `x*I` 可能超过 `u64` 的范围。则编译器必须支持 `u128` 运算，否则此优化难以实现。

也就是说，此优化可能需要编译器支持大整数的运算，适用范围有限。这仍然不够令人满意。

2.3 整除优化：版本三

流程：在版本二的基础上，将 k 值改为 $\lceil \log_2 x \rceil$，然后用相同的方法计算 `x/y` 与 `x%y`。最后追加一次修正操作：若取模结果小于 0，则将取模结果加上 y，将整除结果减去 1。

原理：与版本二同理，列出不等式：

$$0 \leqslant \frac{xI}{2^k} - \frac{x}{y} \leqslant \frac{x(y-1)}{2^k y}$$

当 $k = \lceil \log_2 x \rceil$ 时，有 $\dfrac{x(y-1)}{2^k y} < 1$，则整除运算的误差为 0 或 1，取模运算的误差为 0 或 $-y$。那么在取模结果小于 0 时进行一次修正即可。

评价：对于 $y \geqslant 3$ 的情形，此优化中的 I 只需用与 x 相同的数据类型存储，不需要进行更大范围的整数运算。（而 $y = 1, 2$ 时其实也无需优化）

代价则是：即使只计算整除，为了判断是否要修正误差，也需要额外计算取模。即整除运算所需的乘法次数由 1 变为 2，性能有所下降。

事实上，C/C++ 编译器采取了更高效的改进方案，使整除仍只需执行 1 次乘法。不过相关的数学推导较为繁琐，所以本文将不会对其进行介绍。

3 Barrett 模乘算法：性能测试

选取除数 $y = 101$，使用如下程序连续执行 `u32` 除法共 10^8 次：

```
#include <bits/stdc++.h>
typedef unsigned int u32;
typedef unsigned long long u64;
u32 x=1e8,y=101,n=1e8,I1=-1u/y+1; u64 I2=-1ull/y+1;
u32 mydiv(u32 x) {
    u32 q=(u64)x*I1>>32;
    if (x<q*y) --q;
    return q;
}
int main() {
    int t0=clock(); while (n--) /* B */
    int t1=clock(); printf("%u\n",x);
    printf("%.0lfms\n",1000.*(t1-t0)/CLOCKS_PER_SEC);
    return 0;
}
```

测试结果如下：

编号	说明	B 处语句	耗时
0	乘法对照	`x+=x*y;`	83 ms
1	普通整除	`x+=x/y;`	729 ms
2	手写优化 1	`x+=mydiv(x);`	278 ms
3	编译器优化	`x+=x/101;`	251 ms
4	手写优化 2	`x+=(__int128)x*I2>>64;`	139 ms

表 2: 不同实现下整除运算执行 10^8 次的平均耗时

可以看到，整除运算的性能在优化后得到了十分显著的提升。其中：

- 第 2 种手写优化速度最快，但需要编译器支持足够大的数据类型，适用范围较小。
- 第 1 种手写优化的适用范围、速度都与编译器优化接近。某些场合下，编译器优化不会被触发（例如表中第 1 组），此时可以通过手写优化来达到类似的效果。

4 结论

在编程语言的各种基础运算中，整除、取模运算的开销尤其高昂。

作为注重性能的语言，C 和 C++ 内置了编译优化，基于 Barrett 模乘算法，将除数为常量的除法转化为乘法来计算，使其速度得到大幅提升。

对于编码者而言，Barrett 模乘算法原理简单、代码难度低，必要时可以手动实现，是一种性价比高、值得学习了解的技术。

参考文献

[1] Henry S. Warren. *Hacker' s Delight, Second Edition.* Addison-Wesley Professional, September 2012.