递归算法性能优化——内联汇编分析

黎先波 201511210923
北京师范大学信息科学与技术学院 北京 100875

摘要 当代的计算机性能相对于上个世纪的计算机性能来说已经取得了具大的进步，且随着越来越高效的算法，计算机的发展已经对我们的生活产生了重大的影响。但是对于这个信息大爆炸的时代，对越来越高性能的算法有了更多的需求。递归算法在计算机科学中是指一种通过重复将问题分解为同类的子问题而解决问题的方法，这种方式被用于解决很多的计算机问题且被广泛应用。但同时该算法在面对庞大的数据时任然乏力，本文利用斐波那契数列来从内联汇编的角度尝试分析提高该算法的性能。

关键词 递归算法；性能；汇编优化；斐波那契数列

Recursive Algorithm Performance Optimization——Inline Assembly

Xianbo Li 201511210923
College of Information Science and Technology, Beijing Normal University, Beijing 100875

Abstract: Contemporary computer performance has made great strides relative to the computer performance of the last century, and as more and more efficient algorithms, the development of computers have had a significant impact on our lives. But for the age of this big bang, there is more demand for increasingly higher performance algorithms. Recursive algorithm in computer science refers to a method of solving a problem by repeatedly breaking the problem into similar sub-problems, which is used to solve many computer problems and is widely used. But at the same time, the algorithm is still weak in the face of huge data. This paper uses *fibonacci sequence* to try to analyze and improve the performance of the algorithm from inline assembly.

Keywords: *Recursive Algorithm; Performance; Assembly Optimization; Fibonacci Sequence*

1. 引言

这个学期学了基于 x8086 的汇编语言程序设计，希望能够将这个运用起来，同时刚好在老师的协助下，以及之前已经学过了数据结构、算法以及 C 和 C++，就做了这样一个选题。且刚好在期末展示的时候做的是关于这方面的汇报，因此希望能结合其他几位同学讲的尝试用汇编来优化这个算法。而且在这之前用 Java 编写这个算法的时候，当出现较大的数的时候就会出现短暂的“延时”情况且计算效率不是很好。

斐波纳契数列以意大利数学家雷纳尔多·波纳契（也称为斐波纳契）的名字命名，可在多个学科中找到它的应用，比如数学、建筑学、几何学、计算机科学和自然中。它是从 0, 1 对（或 1, 1）开始的整数数列。数列中的后一个数是前两个数的和。用数学中的项来表示就是，斐波纳契数列 (F_n) 定义由与前两项（F_0、F_1 或 F_1、F_2）的递归关系来定义，如下所示：

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{(n-1)} + F_{(n-2)} \quad (n>1)$$

或者 ①

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{(n-1)} + F_{(n-2)} \quad (n>2)$$

当然，fibonocci 数列还有一种增强形式，即负的 fibonocci 数列。在本文中主要讨论的是非负的 fibonocci 数列（如下表所示，列出了前几项）：

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}
0	1	1	2	3	5	8	13	21	34	55	89

为了比较性能的好坏，故采用以下四种方法来实现 fibonocci 数列。

- 内联汇编方法
- 递归算法
- 迭代实现
- 优化的矩阵乘方算法

在实验中，采用大量的循环次数来延长执行时间，以便正确地记录运行时间。最后的结果表明，采用内联汇编的方式的运行效率要优于其他三种方法。下面就来说明一下实现的过程。

2. 算法过程分析

2.1 递归算法

根据 fibonocci 数列的递归定义（①），可以得到计算第 n 个 fibonocci 数的递归算法是一种直观的体现，可以得到如下的伪指令。

```
Fib(n):
if n in range [0,1]
return n
else
return Fib(n-2)+Fib(n-1)
end if
```

②

按照该递归的定义，该算法的运行的运行时间复杂度可表示为：

$$\begin{aligned} T(n) &= T(n-1)+T(n-2)+\theta(1) \\ &>= 2T(n-2) \\ &= \theta(2^{\frac{n}{2}}) \end{aligned}$$

③

从中可以看出来，这样实现的复杂度是**指数级**的。实现算法：

```
long long fibo_r(int n)
{
        return n<=1?n:fibo_r(n-1)+fibo_r(n-1);
}
```

④

根据这个算法，可以得到如下的递归树：

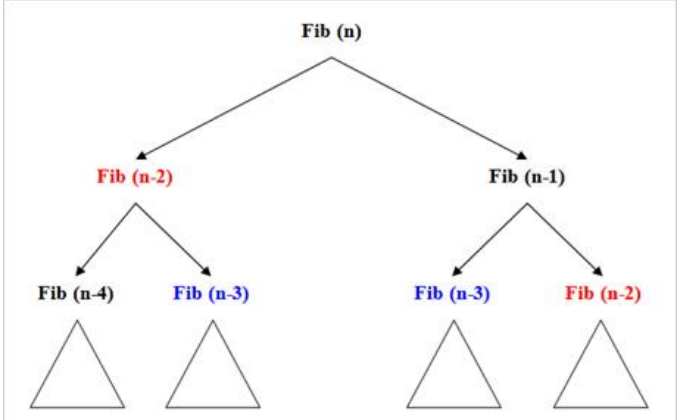


图 1 fibonocci 递归树

从这个递归树中我们可以看出来，有一些表达式经过了多次的计算（红色和蓝色的部分）。由此可以看出这是造成递归算法的时间复杂度为指数级的主要原因。当然这个可以采用动态编程的方法进行解决，但是这不在我们讨论的范围内，我们只是希望通过这个方法来进行比较。

2.2 迭代算法

迭代算法类似于反向遍历法，从索引 0 一直索引到 n 的整个数列。在这个过程中，当前的 fibonocci 数被前两个数的和给更新了。但是在迭代的过程中却没有记住计算的值。如图表示了迭代算法的可视化：

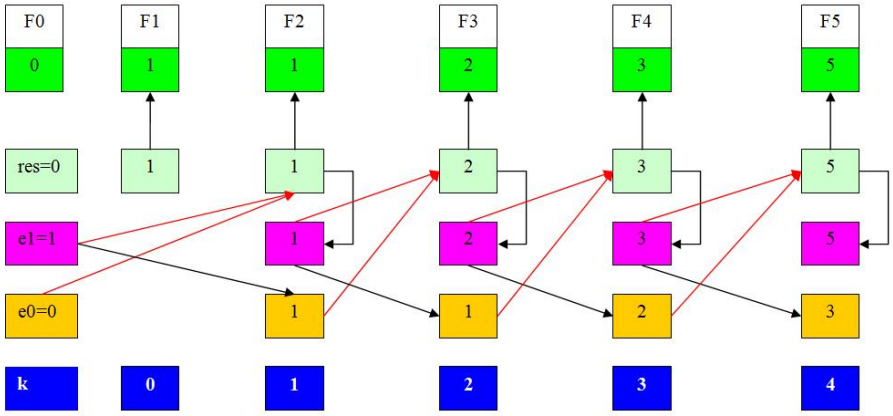


图 2 迭代算法可视化

据此，迭代算法的算法描述如下：

```
long long fibo_i(int n)
{
        int  k;
        long long e0=0,e1=1,res=0;
        for(k=0;k<=n;k++){
                if(k<=1)
                        res=k;
                else{
                        res=e0+e1;
                        e0=e1;
                        e1=res;
                }
        }
        return res;
}
```

⑤

由此可以看出来，迭代算法在执行过程中仅迭代了一次，不会调用递归函数，表明时间复杂度是**线性**的。

2.3 优化的矩阵乘方算法

该算法对于斐波纳契数的计算只有**对数级**的时间复杂度。实现过程如下。对于（①）来说，可以用如下的矩阵来表示：

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$

进一步的，可以推导出

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix} = \ldots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} f(2) \\ f(1) \end{bmatrix}$$

即第 n 个 fibonocci 数将是矩阵$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$扩展到(n-2)乘方后的结果矩阵元素(0 0)。用算法描述如下：

```
void mul(long long M[2][2], long long N[2][2]
{
        long long a=M[0][0]*N[0][0]+M[0][1]*N[1][0];
        long long b=M[0][0]*N[0][1]+M[0][1]*N[1][1];
        long long c=M[1][0]*N[0][0]+M[1][1]*N[1][0];
        long long d=M[1][0]*N[0][1]+M[1][1]*N[1][1];

        M[0][0]=a;
        M[0][1]=b;
        M[1][0]=c;
        M[1][1]=d;
}
```

⑥

为了将矩阵 A 扩展到 n 次方，一种直观的算法将具有线性的运行时间，因为它必须调用乘法运算 (n-1) 次。

$$A^n = AA...A$$

这可以使用以下技术进一步优化：

$$A^n = A^{(n/2)}A^{(n/2)}$$

因此，无需将矩阵 A 扩展到 n 次方，该算法将它扩展到 (n/2) 次方并将 $A^{(n/2)}$ 与自身固定次数的乘法操作。因此得到如下的矩阵乘方运算：

```
void power(long long M[2][2], int n)
{
        if ( n == 0 || n == 1) return;?
        long long A[2][2] = {{1,1},{1,0}};?

        power(M, n/2);?
        mul(M, M);
        if (n%2 != 0)?
                mul(M, A);
}
```

⑦

通过在 n/2 上调用乘方函数，输入的大小会每次减半。相应地，复杂度与**对数**时间有关。下面显

示了一种使用优化的矩阵乘方方法实现。

```
long long fib_m(int n)
{
        long long A[2][2] = {{1,1},{1,0}};
        if (n == 0)return 0;
        power(A, n-1);?
        return A[0][0];
}
```

⑧

以上就是优化的矩阵算法，时间复杂度是**对数**级别的，理论上来说最优的算法。

2.4 内联汇编实现

内联汇编实现将基于迭代算法。从图 2 中显示的可视化形式可以看到，迭代算法计算第 n 项斐波纳契数的方法是将它之前的两项(n-2)和(n-1)相加。(n-2)项在此计算后可丢弃，因为后续计算不会再使用它。据此可用内联汇编方法来实现。

内联汇编实现将两个最新的斐波纳契数保留在两个单独的寄存器中。在每次迭代时，该算法都会计算最新的项的和，然后将计算的值存储在保存较小的项的寄存器中。这将实际覆盖后一项。因为未来的计算不需要较小的项，所以删除此值是可接受的。因此较大的项是这次迭代的结果。图 3 显示了内联汇编方法的可视化表示。

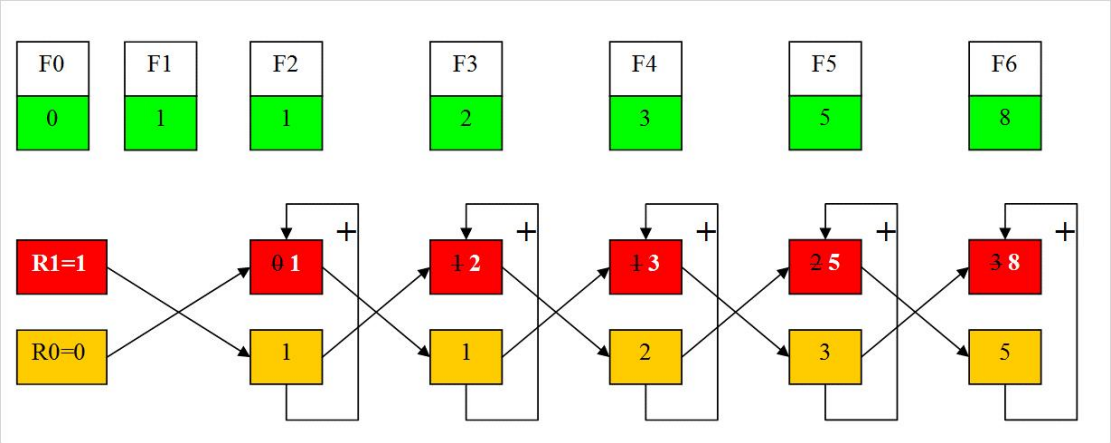


图 3 内联汇编方法可视化

从这个图可以看出来，每次迭代出的结果将存储在红色的寄存器中。可通过如下伪代码来描述：

```
Fibo(n):
        load F0 to register 0
        load F1 to register 1
        for k in range [2, 3,...,n]
                swap the values in two registers
                sum up the values in two register
                store the sum to register 1
        end for
        return register 1
```

⑨

据⑨我们可以计算 n 个 fibonocci 数的实际内联汇编代码如下所示：

```
long long fibo_asm(int n)
{
    n = n - 2;
    int myCount;//保存的fibonocci数
    __asm {
        mov ecx,n//保存所需的斐波纳契数的索引,也是要执行的迭代次数
        mov eax,0//在值eax和ebx中分别加载0和1,其中eax保存计算的数
        mov ebx,1
        add eax,ebx
        L1:
        mov edx,eax                                          ⑨
            mov eax,ebx
            mov ebx,edx
            mov edi,eax
            mov eax,edi
            add si,4
            add eax,ebx
            loop L1
        mov myCount,eax
    }
    return myCount;
}
```

在以上可以看到，首先是用两个寄存器 eax 和 ebx 初始为 1,0，默认为刚开始的两项，其中 eax 最后返回的是计算出的第 n 项的 fibonocci 的值。因为采用的迭代算法，由前面⑤可知，对 fibonocci 数列仅迭代一次，因此时间复杂度和迭算法一样，为**线性级**。

2.5 性能预期

在上面给出的 4 种实现中，具有对数时间复杂度的矩阵乘方算法在性能上排名第一。迭代算法和内联汇编实现其次，具有线性时间复杂度。递归算法由于其指数时间复杂度而性能最差。

运行时复杂度分析

方法	递归算法	迭代算法	优化的矩阵算法	内联汇编
复杂度	指数	线性	对数	线性

因此基于运行时分析，预计在采样大小足够大时，实际性能与复杂度呈正比。接下来进行试验分析。

3. 实验测试与结果

3.1 实验测试

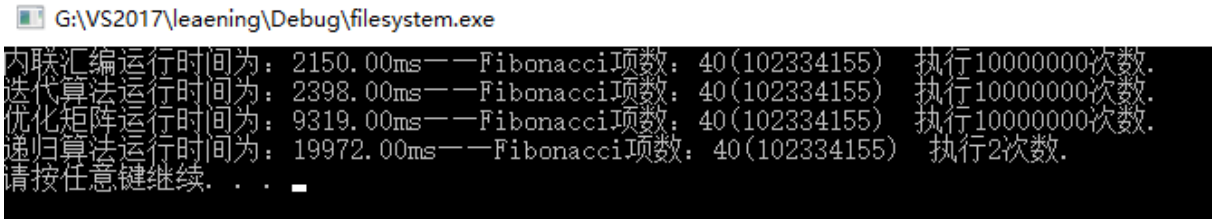
如前面所说的，对于每个算法实现都可以封装在单独的函数当中。进行程序测试的时候主要是看每个函数执行的时间来判断执行的效率，因此使用到了两个变量，startTime 、endTime ，用来分别记录每个函数开始时间和结束时间。同时使用了 4 变量来保存每个函数的运行时间，runningTimeAsm 、runningTimeI 、runningTimeR 、runningTimeM ，分别对应内联汇编函数的运行时间，迭代算法的运行时间，递归算法的运行时间，优化矩阵的运行时间。

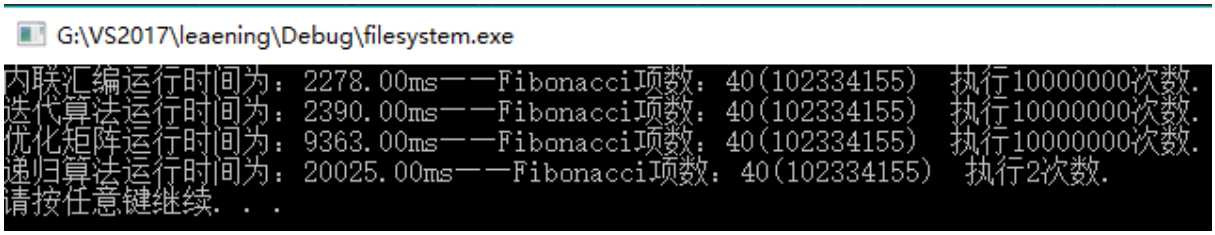
运行时间的计算方法由如下程序给出（完整程序请看附录）：

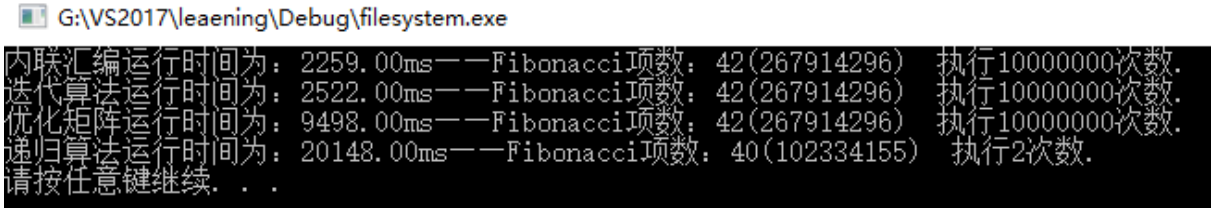
```
clock_t startTime = clock();
while(myCount--){resultAsm=fibo_asm(11);}
 clock_t endTime = clock();
runningTimeAsm=((float)(endTime-startTime)*1000)/COLCKS_PER_SEC;
```

除此以外，考虑到实际测试的环境，每个函数的实现都用 40~60 项执行10,000,000 次运算。采用大量的循环次数用于延长执行时间，以便正确记录运行时间。另外，由于大于斐波纳契数列中第 100 个数的项超出了整数数据类型长度，而第 42 项具有 32 位整数的斐波纳契计算的上限。反之递归函数将使用小得多的参数来进行测试，因为它具有指数时间复杂度，应该选择较小的参数来确保程序在其他方法的时间范围内完成，所以只用它计算斐波纳契数列的第 40 项两次。实验结果请看下面

3.2 实验结果







从图中可以明显地看到，在计算最大的斐波纳契数时，内联汇编实现优于其他所有方法。相比优化的矩阵算法和迭代算法性能都要好。这一性能优势源于内联汇编实现较小的资源占用。同时从这个结果中也可以看出，内联汇编实现生成的紧凑代码的实际速度可能超过某种更快的算法的性能。这是因为使用内联汇编进一步提高了性能，因此这为超越最佳算法的性能提供了一个示例。

在计算斐波纳契数列的特殊情况下，内联汇编方法的性能优于其他所有算法，即使已证明理论上速度将会更慢。这个示例证明，理论性能可能不切实际。适当地使用内联汇编来调优最注重性能的代码节，这是超越算法所提供的默认速度的一条途径。但是在大多数情况下，使用编译器所提供的适当的优化水平可能是最佳选择。

4. 参考资料

https://www.ibm.com/developerworks/linux/library/l-performance-of-inline-assembly-trs/fibonacci-anhtran.c

https://msdn.microsoft.com/zh-cn/library/5sds75we.aspx

http://blog.csdn.net/hkbyest/article/details/1684384

https://www.cnblogs.com/stacktrace/p/5142470.html

http://blog.csdn.net/xpplearnc/article/details/53894048

https://chenqx.github.io/2014/09/29/Algorithm-Recursive-Programming/

《汇编语言：基于 x86 处理器》 【美】kip Irvine 【译者】贺莲 龚奕利 机械工业出版社 2016 年 4 月第一版第一次印刷

5. 附录

完整程序如下所示：

```
#include<iostream>
#include<ctime>
#include<iomanip>

using namespace std;

/*递归算法*/
long long fibo_r(int n)
{
if (n == 0)return 0;
if (n == 1)return 1;
return fibo_r(n - 1) + fibo_r(n - 2);
}

/*内联汇编*/
unsigned int fibo_asm(int n)
{
n = n - 2;
unsigned int myCount;//保存的 fibonocci 数
__asm {
mov ecx,n//保存所需的斐波纳契数的索引,也是要执行的迭代次数
mov eax,0//在值 eax 和 ebx 中分别加载 0 和 1,其中 eax 保存计算的数
mov ebx,1
add eax,ebx
L1:
mov edx,eax
mov eax,ebx
mov ebx,edx
mov edi,eax
mov eax,edi
add si,4
add eax,ebx
loop L1
mov myCount,eax
}
return myCount;
}

/*迭代算法*/
long long fibo_i(int n)
{
int k;
long long e0 = 0, e1 = 1, res = 0;
for (k = 0; k <= n; ++k) {
if (k <= 1)
```

```
res = k;
else {
res = e0 + e1;
e0 = e1;
e1 = res;
}
}
return res;
}

void mul(long long M[2][2], long long N[2][2])
{
long long a = M[0][0] * N[0][0] + M[0][1] * N[1][0];
long long b = M[0][0] * N[0][1] + M[0][1] * N[1][1];
long long c = M[1][0] * N[0][0] + M[1][1] * N[1][0];
long long d = M[1][0] * N[0][1] + M[1][1] * N[1][1];

M[0][0] = a;
M[0][1] = b;
M[1][0] = c;
M[1][1] = d;
}

void power(long long M[2][2], int n)
{
if (n == 0 || n == 1) return;
long long A[2][2] = { {1,1},{1,0} };
power(M, n / 2);
mul(M, M);
if (n % 2 != 0) mul(M, A);
}

/*优化矩阵算法*/
long long fibo_m(int n)
{
long long A[2][2] = { { 1,1 },{ 1,0 } };
if (n == 0) return 0;
power(A, n - 1);
return A[0][0];
}

int main()
{
clock_t startTime,endTime;
float runningAsm, runningI, runningR, runningM;
long long resultI = 0, resultR = 0, resultM = 0, myCount;
unsigned int resultASM = 0;
```

```
        int limit = 42;
        long long loopCount = 10000000;

        /*内联汇编函数测试*/
        myCount = loopCount;
        startTime = clock();
        while (myCount--) { resultASM = fibo_asm(limit); }
        endTime = clock();
        runningAsm = ((float)(endTime - startTime)) * 1000 / CLOCKS_PER_SEC;

        /*迭代函数测试*/
        myCount = loopCount;
        startTime = clock();
        while (myCount--) { resultI = fibo_i(limit); }
        endTime = clock();
        runningI = ((float)(endTime - startTime)) * 1000 / CLOCKS_PER_SEC;

        /*优化矩阵函数测试*/
        myCount = loopCount;
        startTime = clock();
        while (myCount--) { resultM = fibo_m(limit); }
        endTime = clock();
        runningM = ((float)(endTime - startTime)) * 1000 / CLOCKS_PER_SEC;

        /*递归函数测试*/
        int rCount = 2;
        int rLimit = 40;
        startTime = clock();
        while (rCount--) { resultR = fibo_r(rLimit); }
        endTime = clock();
        runningR = ((float)(endTime - startTime)) * 1000 / CLOCKS_PER_SEC;

        cout << "内联汇编运行时间为：" << setiosflags(ios::fixed) << setprecision(2) << runningAsm <<
"ms";
        cout << "——Fibonacci 项数：" << limit << "(" << resultASM << ")" << "  执行" << loopCount <<
"次数." << endl;

        cout << "迭代算法运行时间为：" << setiosflags(ios::fixed) << setprecision(2) << runningI << "ms";
        cout << "——Fibonacci 项数：" << limit << "(" << resultI << ")" << "  执行" << loopCount << "次
数." << endl;

        cout << "优化矩阵运行时间为：" << setiosflags(ios::fixed) << setprecision(2) << runningM <<
"ms";
        cout << "——Fibonacci 项数：" << limit << "(" << resultM << ")" << "  执行" << loopCount << "次
数." << endl;
```

```
        cout << "递归算法运行时间为：" << setiosflags(ios::fixed) << setprecision(2) << runningR << "ms";
        cout << "——Fibonacci 项数：" << rLimit << "(" << resultR << ")" << "    执行" << 2 << "次数." <<
endl;

        system("pause");
        return 0;
        }
```