c/c++测试函数的运行时间（八种方法）

目前，存在着各种计时函数，一般的处理都是先调用计时函数，记下当前时间tstart，然后处理一段程序，再调用计时函数，记下处理后的时间tend，再tend和tstart做差，就可以得到程序的执行时间，但是各种计时函数的精度不一样.下面对各种计时函数，做些简单记录.

```
void foo()
{
    long i;
    for (i=0;i<100000000;i++)
    {
        long a= 0;
        a = a+1;
    }
}
```

方法1,time()获取当前的系统时间，返回的结果是一个time_t类型，其实就是一个大整数，其值表示从CUT（Coordinated Universal Time）时间1970年1月1日00:00:00（称为UNIX系统的Epoch时间）到当前时刻的秒数.

```
void test1()
{
    time_t start,stop;
    start = time(NULL);
    foo();//dosomething
    stop = time(NULL);
    printf("Use Time:%ld\n",(stop-start));
}
```

方法2,clock()函数返回从“开启这个程序进程”到“程序中调用clock()函数”时之间的CPU时钟计时单元（clock tick）数，在MSDN中称之为挂钟时间（wal-clock）常量CLOCKS_PER_SEC，它用来表示一秒钟会有多少个时钟计时单元。

```
void test2()
{
    double dur;
    clock_t start,end;
    start = clock();
    foo();//dosomething
    end = clock();
    dur = (double)(end - start);
    printf("Use Time:%f\n",(dur/CLOCKS_PER_SEC));
}
```

方法3,timeGetTime()函数以毫秒计的系统时间。该时间为从系统开启算起所经过的时间,是windows api，需要头文件windows.h

```
void test3()
{
    DWORD t1,t2;
    t1 = timeGetTime();
    foo();//dosomething
    t2 = timeGetTime();
    printf("Use Time:%f\n",(t2-t1)*1.0/1000);
}
```

方法4,QueryPerformanceCounter()这个函数返回高精确度性能计数器的值,它可以以微妙为单位计时.但是QueryPerformanceCounter()确切的精确计时的最小单位是与系统有关的,所以,必须要查询系统以得到QueryPerformanceCounter()返回的嘀哒声的频率.QueryPerformanceFrequency()提供了这个频率值,返回每秒嘀哒声的个数.

```
void test4()
{
    LARGE_INTEGER t1,t2,tc;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);
    foo();//dosomething
    QueryPerformanceCounter(&t2);
    printf("Use Time:%f\n",(t2.QuadPart - t1.QuadPart)*1.0/tc.QuadPart);
}
```

方法5,GetTickCount返回（retrieve）从操作系统启动到现在所经过（elapsed）的毫秒数，它的返回值是DWORD

```
void test5()
{
    DWORD t1,t2;
    t1 = GetTickCount();
    foo();//dosomething
    t2 = GetTickCount();
    printf("Use Time:%f\n",(t2-t1)*1.0/1000);
}
```

方法6,RDTSC指令，在Intel Pentium以上级别的CPU中，有一个称为“时间戳（Time Stamp）”的部件，它以64位无符号整型数的格式，记录了自CPU上电以来所经过的时钟周期数。由于目前的CPU主频都非常高，因此这个部件可以达到纳秒级的计时精度。这个精确性是上述几种方法所无法比拟的.在Pentium以上的CPU中，提供了一条机器指令RDTSC（Read Time Stamp Counter）来读取这个时间戳的数字，并将其保存在EDX:EAX寄存器对中。由于EDX:EAX寄存器对恰好是Win32平台下C++语言保存函数返回值的寄存器，所以我们可以把这条指令看成是一个普通的函数调用，因为RDTSC不被C++的内嵌汇编器直接支持，所以我们要用_emit伪指令直接嵌入该指令的机器码形式0X0F、0X31

```
inline unsigned __int64 GetCycleCount()
{
    __asm
    {
        _emit 0x0F;
        _emit 0x31;
    }
}

void test6()
{
    unsigned long t1,t2;
    t1 = (unsigned long)GetCycleCount();
    foo();//dosomething
    t2 = (unsigned long)GetCycleCount();
    printf("Use Time:%f\n",(t2 - t1)*1.0/FREQUENCY);    //FREQUENCY指CPU的频率
}
```

方法7,gettimeofday() linux环境下的计时函数，int gettimeofday (struct timeval * tv , struct timezone * tz),gettimeofday()会把目前的时间有tv所指的结构返回，当地时区的信息则放到tz所指的结构中.

```
//timeval结构定义为:
struct timeval{
long tv_sec; /*秒*/
long tv_usec; /*微秒*/
};
//timezone 结构定义为:
struct timezone{
int tz_minuteswest; /*和Greenwich 时间差了多少分钟*/
int tz_dsttime; /*日光节约时间的状态*/
};
```

```
void test7()
{
    struct timeval t1,t2;
    double timeuse;
    gettimeofday(&t1,NULL);
    foo();
    gettimeofday(&t2,NULL);
    timeuse = t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.tv_usec)/1000000.0;
    printf("Use Time:%f\n",timeuse);
}
```

方法8,linux环境下，用RDTSC指令计时.与方法6是一样的.只不过在linux实现方式有点差异.

```
#if defined (__i386__)
static __inline__ unsigned long long GetCycleCount(void)
{
        unsigned long long int x;
        __asm__ volatile("rdtsc":"=A"(x));
        return x;
}
#elif defined (__x86_64__)
static __inline__ unsigned long long GetCycleCount(void)
{
        unsigned hi,lo;
        __asm__ volatile("rdtsc":"=a"(lo),"=d"(hi));
        return ((unsigned long long)lo)|(((unsigned long long)hi)<<32);
}
#endif

void test8()
{
        unsigned long t1,t2;
        t1 = (unsigned long)GetCycleCount();
        foo();//dosomething
        t2 = (unsigned long)GetCycleCount();
        printf("Use Time:%f\n",(t2 - t1)*1.0/FREQUENCY); //FREQUENCY  CPU的频率
}
```

总结，方法1,2,7,8可以在linux环境下执行，方法1,2,3,4,5,6可以在windows环境下执行.其中，timeGetTime()和GetTickCount()的返回值类型为DWORD，当统计的毫妙数过大时，将会使结果归0，影响统计结果.
测试结果，windows环境下，主频为1.6GHz，单位为秒.

1 Use Time:0
2 Use Time:0.390000
3 Use Time:0.388000
4 Use Time:0.394704
5 Use Time:0.407000
6 Use Time:0.398684