

74 BENCHMARKING in C++ (how to measure performance)

1. 基准测试

这里的基准测试不仅仅是你用来对代码进行基准测试的工具，如果你想衡量一段C++代码的性能，而这段代码本身就需要你正确地去写。

有多种方法可以测量C++代码的性能，这里只讲述了Cherno使用的方法。

首先要写一些我们想测试的代码：

```
C++

int main()
{
    int value = 0;
    for (int i = 0; i < 1000000; i++)
        value += 2;

    std::cout << value << std::endl;    // 2000000

    __debugbreak(); // VS打断点专用语法
}
```

现在想要分析我们的代码有多快，可以创建一个简单的、有作用域的计时（可参考[63 Timing in C++](#)）

```

#include <chrono>

class Timer
{
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> m_StartTimepoint;
public:
    Timer()
    {
        m_StartTimepoint = std::chrono::high_resolution_clock::now();
    }

    ~Timer()
    {
        Stop();
    }

    void Stop()
    {
        auto endTimepoint = std::chrono::high_resolution_clock::now();

        auto start = std::chrono::time_point_cast<std::chrono::microseconds>
(m_StartTimepoint).time_since_epoch().count();

        auto end = std::chrono::time_point_cast<std::chrono::microseconds>
(endTimepoint).time_since_epoch().count();

        auto duration = end - start;
        double ms = duration * 0.001;

        std::cout << duration << "us (" << ms << "ms)\n";
    }
};

int main()
{
    int value = 0; // 挪到作用域外, 保证可以打印出来
    {
        Timer timer;
        for (int i = 0; i < 1000000; i++)
            value += 2;
    }
    std::cout << value << std::endl;
    __debugbreak();
}

```

打印出计时:

```
value += 2;

std::cout << value <<
__debugbreak();
```

1887us (1.887ms)
2000000

这里要注意的是，你要确保自己正在测量的东西是实际上被编译的代码，因为Release模式下编译器会优化汇编指令，这个例子中只会记录打印变量1E8480h（两百万）所需的时间（由于打印在作用域外甚至什么都没统计）：

```
Timer timer;
for (int i = 0; i < 1000000;
    value += 2;
}

std::cout << value
__debugbreak();
```

0us (0ms)
2000000

而不是你想统计的加一百万次的用时：

```
std::cout << value << std::endl;
001210E7 push offset std::endl<char,std::char_traits<char> > (0121420h)
for (int i = 0; i < 1000000; i++)
    value += 2;
}
001210EC mov dword ptr [ebp-4],0FFFFFFFh
std::cout << value << std::endl;
001210F3 mov ecx,dword ptr [__imp_std::cout (0123064h)]
001210F9 push 1E8480h
001210FE call dword ptr [__imp_std::basic_ostream<char,std::char_traits<char>
```

2. 智能指针的性能对比

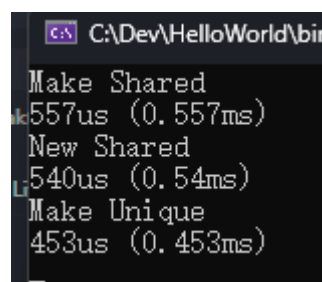
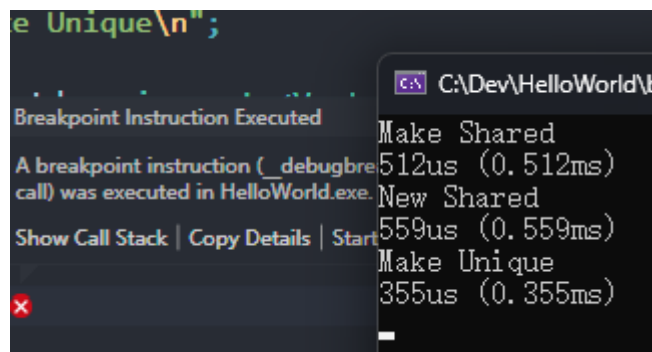
```

int main()
{
    struct Vector2
    {
        float x, y;
    };

    std::cout << "Make Shared\n";
    {
        std::array < std::shared_ptr<Vector2>, 1000> sharedPtrs;
        Timer timer; // 不统计进创造数组的时间
        for (int i = 0; i < sharedPtrs.size(); i++)
            sharedPtrs[i] = std::make_shared<Vector2>();
    }
    std::cout << "New Shared\n";
    {
        std::array < std::shared_ptr<Vector2>, 1000> sharedPtrs;
        Timer timer;
        for (int i = 0; i < sharedPtrs.size(); i++)
            sharedPtrs[i] = std::shared_ptr<Vector2>(new Vector2());
    }
    std::cout << "Make Unique\n";
    {
        std::array < std::unique_ptr<Vector2>, 1000> sharedPtrs;
        Timer timer;
        for (int i = 0; i < sharedPtrs.size(); i++)
            sharedPtrs[i] = std::make_unique<Vector2>();
    }
    __debugbreak();
}

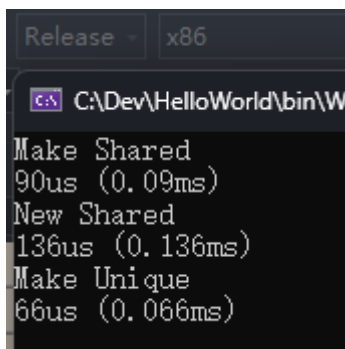
```

测量两次 `shared_ptr` 和 `unique_ptr` 的性能对比:



和我们所料的差不多，`unique_ptr` 比 `shared_ptr` 花的时间少，但是 `make_shared` 和 `new` 所差的时间差不多。这里有件很重要的事，我们实际上是在Debug模式下分析的，因为它有很多额外的安全措施，需要时间而对测量性能不是很好。

切换到Release模式，可以发现 `make_shared` 明显比 `new` 快：



```
Release x86
C:\Dev\HelloWorld\bin\W
Make Shared
90us (0.09ms)
New Shared
136us (0.136ms)
Make Unique
66us (0.066ms)
```

所以一定要确保你所分析的代码，是在Release时真正有意义的，因为你不会在Debug时发布代码。