81 VISUAL BENCHMARKING in C++

如何可视化地测量性能。前置知识：74 BENCHMARKING in C++ (how to measure performance)

1. Basic benchmark

C++

```cpp
#include "pch.h"
#include <cmath>
#include <chrono>

class Timer // 基准测试，见74课
{
public:
    Timer(const char* name)
        : m_Name(name), m_Stopped(false)
    {
        m_StartTimepoint = std::chrono::high_resolution_clock::now();
    }

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

        long long start = std::chrono::time_point_cast<std::chrono::milliseconds>
(m_StartTimepoint).time_since_epoch().count();
        auto end = std::chrono::time_point_cast<std::chrono::milliseconds>
(endTimepoint).time_since_epoch().count(); // 也是long long

        std::cout << m_Name << ": " << (end - start) << "ms\n";

        m_Stopped = true;
    }

    ~Timer()
    {
        if (m_Stopped = true)
            Stop();
    }

private:
    const char* m_Name;
    std::chrono::time_point<std::chrono::steady_clock> m_StartTimepoint;
    bool m_Stopped;
};

void Function1()
{
    Timer time("Function1");
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << i << std::endl;
}

void Function2()
{
    Timer time("Function2");
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << sqrt(i) << std::endl;
}
```

```
int main()
{
    Function1();
    Function2();
    std::cin.get();
}

// 结果:
/*
HelloWorld#0 ...
HelloWorld#999 ...
Function1: 611ms
HellowWorld#0 ...
Hello World #31.607
Function2: 1037ms
*/
```

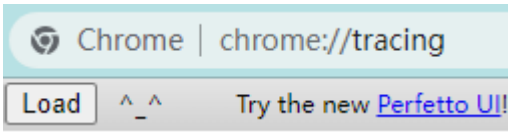
现在结果只是数字，而且浏览寻找很麻烦，于是我们进入Visualization可视化环节：

2. Visualization

我们要进入Google Chrome，我反正猜不到可以用它来做可视化分析...

Chrome自带一些自己的分析工具和其它开发工具，很明显是针对网络应用或网页的，还有一个特别的叫Chome Tracing，它很*barebone*（准系统），很简单而基础，让我们能够可视化我们的分析和堆栈跟踪视图。

输入chrome://tracing即可进入，而且很可能已经装在电脑里了：



它的工作方式是加载一个包含所有数据的json文件，我们的下一步是取所有我们用计时器记录的计时数据，把它放入一个用于Chrome tracing的json格式文件。

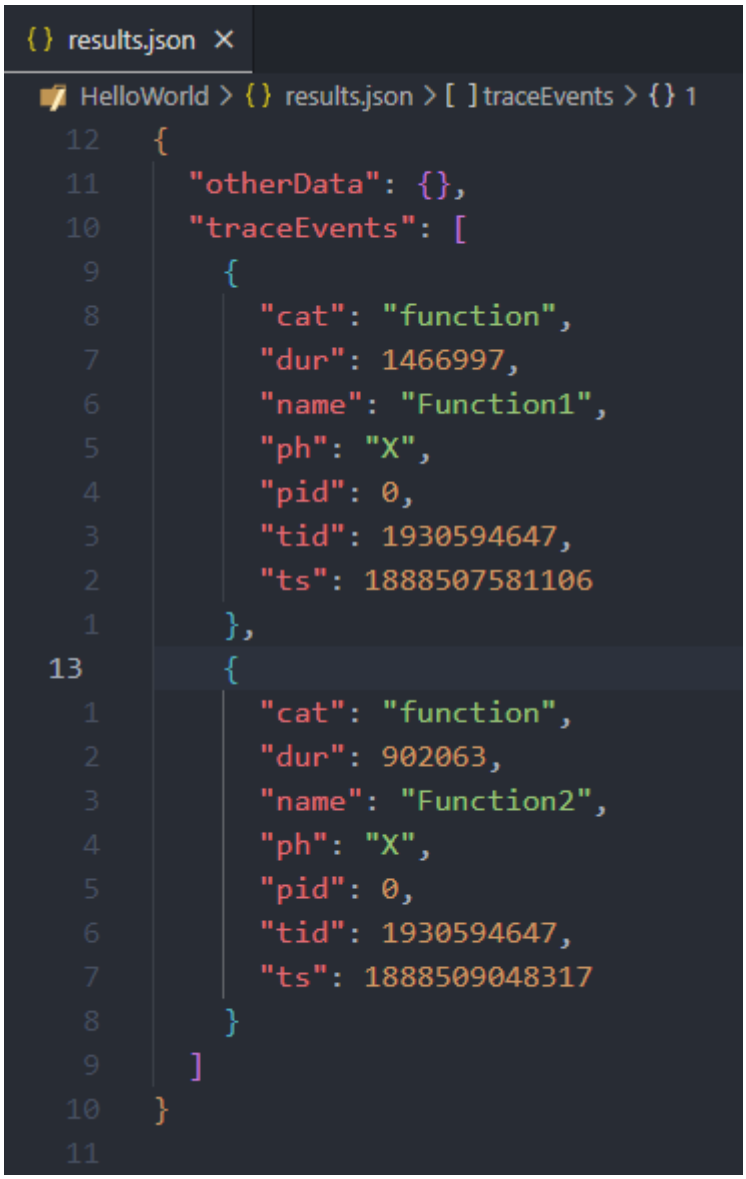
C++

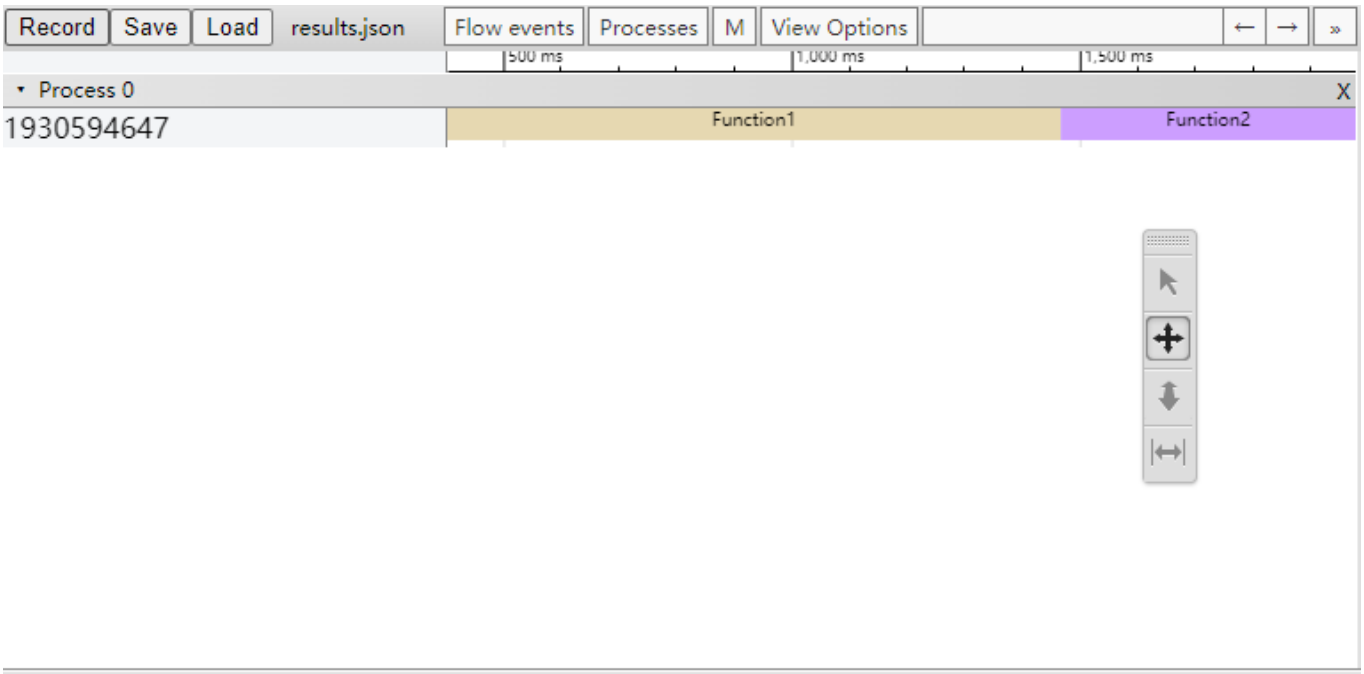
```cpp
// 结构体ProfileResult，用于存储性能测试结果
struct ProfileResult
{
    std::string Name;  // 测试名称
    long long Start, End;  // 测试开始和结束的时间点
    uint32_t ThreadID;  // 执行测试的线程ID
};

// 结构体InstrumentationSession，用于存储一个测试会话的信息
struct InstrumentationSession
{
    std::string Name;  // 会话名称
};

// 类Instrumentor，用于进行性能测试和结果输出
class Instrumentor
{
private:
    InstrumentationSession* m_CurrentSession;  // 当前的测试会话
    std::ofstream m_OutputStream;  // 输出流，用于写入测试结果
    int m_ProfileCount;  // 记录当前会话已完成的测试数量
public:
    // 构造函数
    Instrumentor()
        : m_CurrentSession(nullptr), m_ProfileCount(0)
    {
    }

    // 开始一个新的测试会话
    void BeginSession(const std::string& name, const std::string& filepath =
"results.json")
    {
        m_OutputStream.open(filepath);  // 打开输出文件
        WriteHeader();  // 写入测试结果的头部信息
        m_CurrentSession = new InstrumentationSession{ name };  // 创建新的会话
    }

    // 结束当前的测试会话
    void EndSession()
    {
        WriteFooter();  // 写入测试结果的尾部信息
        m_OutputStream.close();  // 关闭输出文件
        delete m_CurrentSession;  // 删除当前会话
        m_CurrentSession = nullptr;  // 将当前会话指针设置为nullptr
        m_ProfileCount = 0;  // 重置测试数量
    }

    // 将一个测试结果写入输出文件
    void WriteProfile(const ProfileResult& result)
    {
        if (m_ProfileCount++ > 0)
            m_OutputStream << ",";  // 如果已经有测试结果，那么在新的测试结果前添加一个逗号
```

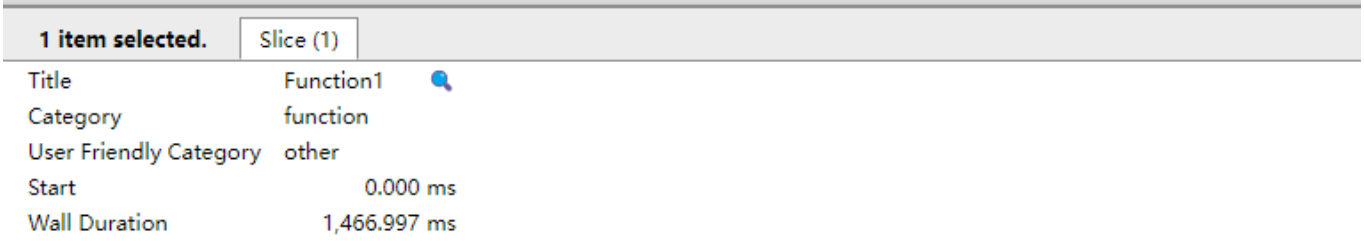
```
            // 处理测试名称中可能存在的双引号字符
            std::string name = result.Name;
            std::replace(name.begin(), name.end(), '"', '\'');

            // 写入测试结果
            // 测试结果的格式是JSON，包含测试名称、开始和结束时间、线程ID等信息
            m_OutputStream << "{";
            m_OutputStream << "\"cat\":\"function\",";
            m_OutputStream << "\"dur\":" << (result.End - result.Start) << ',';
            m_OutputStream << "\"name\":\"" << name << "\",";
            m_OutputStream << "\"ph\":\"X\",";
            m_OutputStream << "\"pid\":0,";
            m_OutputStream << "\"tid\":" << result.ThreadID << ",";
            m_OutputStream << "\"ts\":" << result.Start;
            m_OutputStream << "}";

            m_OutputStream.flush();  // 刷新输出流，确保结果被写入文件
        }

        // 写入测试结果的头部信息
        void WriteHeader()
        {
            m_OutputStream << "{\"otherData\": {},\"traceEvents\":[";
            m_OutputStream.flush();
        }

        // 写入测试结果的尾部信息
        void WriteFooter()
        {
            m_OutputStream << "]}";
            m_OutputStream.flush();
        }

        // 获取Instrumentor的单例
        // 由于这是一个性能测试工具，我们通常只需要一个实例
        static Instrumentor& Get()
        {
            static Instrumentor instance;
            return instance;
        }
    };

    // 类InstrumentationTimer，用于计时和性能测试
    class InstrumentationTimer
    {
    public:
        // 构造函数
        // 在创建对象时开始计时
        InstrumentationTimer(const char* name)
            : m_Name(name), m_Stopped(false)
        {
            m_StartTimepoint = std::chrono::high_resolution_clock::now();
        }
```

```
    // 析构函数
    // 如果计时器没有停止，那么在对象被销毁时自动停止计时，并记录测试结果
    ~InstrumentationTimer()
    {
        if (!m_Stopped)
            Stop();
    }

    // 停止计时，并记录测试结果
    void Stop()
    {
        // 获取当前时间点
        auto endTimepoint = std::chrono::high_resolution_clock::now();

        // 计算开始和结束的时间（单位：微秒）
        long long start = std::chrono::time_point_cast<std::chrono::microseconds>
(m_StartTimepoint).time_since_epoch().count();
        long long end = std::chrono::time_point_cast<std::chrono::microseconds>
(endTimepoint).time_since_epoch().count();

        // 获取当前线程的ID
        uint32_t threadID = std::hash<std::thread::id>{}(std::this_thread::get_id());

        // 将测试结果写入输出文件
        Instrumentor::Get().WriteProfile({ m_Name, start, end, threadID });

        m_Stopped = true;  // 标记计时器已停止
    }
private:
    const char* m_Name;  // 测试名称
    std::chrono::time_point<std::chrono::high_resolution_clock> m_StartTimepoint;  //
计时开始的时间点
    bool m_Stopped;  // 标记是否已经停止计时
};

void Function1()
{
    InstrumentationTimer time("Function1");
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << i << std::endl;
}

void Function2()
{
    InstrumentationTimer time("Function2");
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << sqrt(i) << std::endl;
}

int main()
{
    Instrumentor::Get().BeginSession("Profile"); // 启动会话
    Function1();
    Function2();
```

```
    Instrumentor::Get().EndSession();
    std::cin.get();
}
```

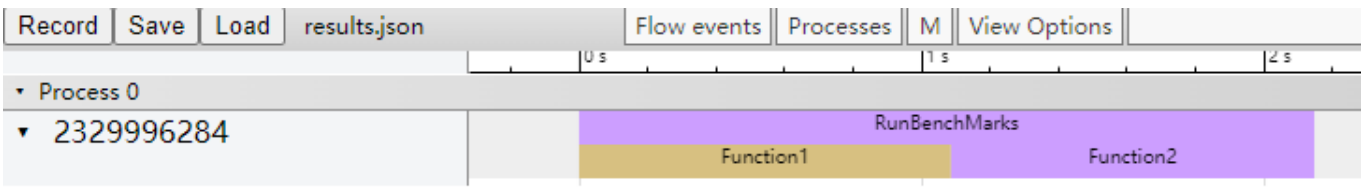






C++

```cpp
void RunBenchmarks()
{
    InstrumentationTimer timer("RunBenchMarks");
    std::cout << "Running Benchmarks...\n";
    Function1();
    Function2();
}

int main()
{
    Instrumentor::Get().BeginSession("Profile");
    RunBenchmarks();
    Instrumentor::Get().EndSession();
    std::cin.get();
}
```



但这个RunBenchmark必须复制粘贴我们调用的每个函数的名称，比较麻烦，另外这样的计时代码不是我们想要在程序中一直运行的东西，应该有一个简单的方法可以关闭这些来减少开销，所以可以写一些宏来解决这两个问题：

```C++
#define PROFILING 1
#if PROFILING
#define PROFILE_SCOPE(name) InstrumentationTimer timer##__LINE__(name);
#define PROFILE_FUNCTION() PROFILE_SCOPE(__FUNCTION__)// 预定义的宏，会返回一个包含当前函数名称的字符串。
#else
#define PROFILE_SCOPE(name)
#endif

void Function1()
{
    PROFILE_FUNCTION();
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << i << std::endl;
}

void Function2()
{
    PROFILE_FUNCTION();
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << sqrt(i) << std::endl;
}

void RunBenchmarks()
{
    PROFILE_FUNCTION();
    std::cout << "Running Benchmarks...\n";
    Function1();
    Function2();
```

> 也就是让预处理器帮我们完成，不用自己输入字符串

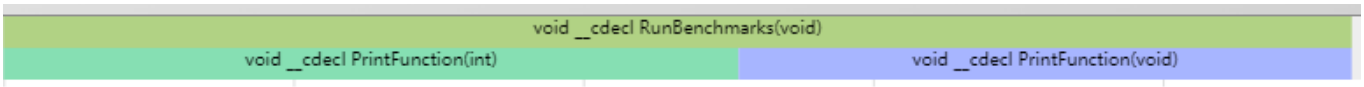
如果我们想要更多信息，比如说有函数重载的情况：

```cpp
void PrintFunction(int value)
{
    PROFILE_FUNCTION();
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << (i + value) << std::endl;
}

void PrintFunction()
{
    PROFILE_FUNCTION();
    for (int i = 0; i < 1000; i++)
        std::cout << "Hello World #" << sqrt(i) << std::endl;
}

void RunBenchmarks()
{
    PROFILE_FUNCTION();
    std::cout << "Running Benchmarks...\n";
    PrintFunction(value: 1);
    PrintFunction();
}
```

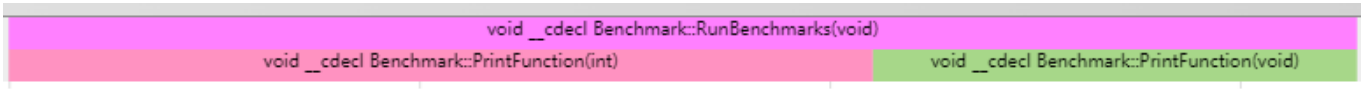
这样两个函数打印出来的名字是相同的，我们想要函数签名怎么办？

只需要修改一下宏：

```cpp
#define PROFILE_FUNCTION() PROFILE_SCOPE(__FUNCSIG__)// 预定义的宏，返回一个包含当前函数签名
```



可以添加一个名称空间，__FUNGSIG__会给你全部信息：



ChromeTracing支持的另一个很酷的东西是多线程，前面的完整代码中已加入线程，但是并没有调用，所以：

C++

```cpp
void RunBenchmarks()
{
    PROFILE_FUNCTION();
    std::cout << "Running Benchmarks...\n";
    std::thread a([]() {PrintFunction(1); });
    std::thread b([]() {PrintFunction(); });
    // 最后两个join让这两个线程都完成工作前，不会真正地退出这个Benchmark函数
    a.join();
    b.join();
}
```