# 探索 C++ 中的多线程

迈克尔·阿代索

2019 年 11 月 16 日 警察

评价我: ★★★★★ 5.00/5 (7 票)

C++中的多线程

作为一名开发人员，对性能的追求是我感兴趣的事情，因此作为一项学习练习，我决定尝试并巩固我对多线程的了解。如今，随着我们的 CPU 获得越来越多的内核，它变得更加重要。现代游戏引擎和应用程序使用多个 CPU 内核来保持快速响应。

## 系列文章

- **第 1 部分：探索 C++ 中的多线程**
- 第 2 部分：探索 C++ 中的多线程续。
- 第 3 部分：探索 C++ 中的多线程：加载纹理
- 第 4 部分：探索 C++ 中的多线程：并行化光线追踪

## 设置和基线结果

作为一个测试用例，我决定创建一系列任务，一些小的和一些大的，来模拟不同的工作负载类型。作为一个简单的测试用例，我找到了一种计算 Pi 的方法，并多次运行该方法，具体取决于我希望工作负载有多大。

C++ 复制代码

```
double CalcPi(int n)
{
    double sum = 0.0;
    int sign = 1;
    for (int i = 0; i < n; ++i)
    {
        sum += sign / (2.0 * i + 1.0);
        sign *= -1;
    }
    return 4.0 * sum;
}
```

现在我创建了几个不同的正在运行的作业`CalcPi`并将它们添加到向量或队列中（取决于我正在运行的测试）。我的`CalcPiJob`班级看起来像这样：

C++ 复制代码

```
class CalcPiJob
{
```

```
public:
    CalcPiJob(int iterations)
        : m_iterations(iterations)
    { }

    void DoWork()
    {
        float p = 0.0f;
        for (int i = 0; i < m_iterations; ++i) {
            p += CalcPi(m_iterations);
        }

        p /= m_iterations;
        std::this_thread::sleep_for(std::chrono::milliseconds(Settings::ThreadPause));
    }

private:
    int m_iterations;
};
```

创建一系列不同的工作负载类型如下所示:

C++ 缩小▲ 复制代码

```
std::queue<CalcPiJob*> GetJobsQ()
{
    std::queue<CalcPiJob*> jobQ;
    for (int i = 0; i < Settings::JobCountHigh; ++i)
    {
        jobQ.emplace(new CalcPiJob(Settings::IterationCountHigh));
    }

    for (int i = 0; i < Settings::JobCountMedium; ++i)
    {
        jobQ.emplace(new CalcPiJob(Settings::IterationCountMedium));
    }

    for (int i = 0; i < Settings::JobCountLow; ++i)
    {
        jobQ.emplace(new CalcPiJob(Settings::IterationCountLow));
    }
    return jobQ;
}

std::vector<CalcPiJob*> GetJobVector()
{
    std::vector<CalcPiJob*> jobs;
    for (int i = 0; i < Settings::JobCountHigh; ++i)
    {
        jobs.push_back(new CalcPiJob(Settings::IterationCountHigh));
    }

    for (int i = 0; i < Settings::JobCountMedium; ++i)
    {
        jobs.push_back(new CalcPiJob(Settings::IterationCountMedium));
    }

    for (int i = 0; i < Settings::JobCountLow; ++i)
    {
        jobs.push_back(new CalcPiJob(Settings::IterationCountLow));
    }
    return jobs;
}
```

我还定义了几个常量来提供帮助。

C++ 复制代码

```
struct Settings
{
    enum class Priority : int {
        Low = 0,
        Medium,
        High
    };

    static const int JobCountLow = 120;
    static const int JobCountMedium = 60;
    static const int JobCountHigh = 25;

    static const int ThreadPause = 100;

    static const int IterationCountLow = 5000;
    static const int IterationCountMedium = 10000;
    static const int IterationCountHigh = 20000;

    static const int PrecisionHigh = 100;
    static const int PrecisionMedium = 100;
    static const int PrecisionLow = 100;
};
```

现在作为基线，我遍历所有作业并按DoWork顺序执行。

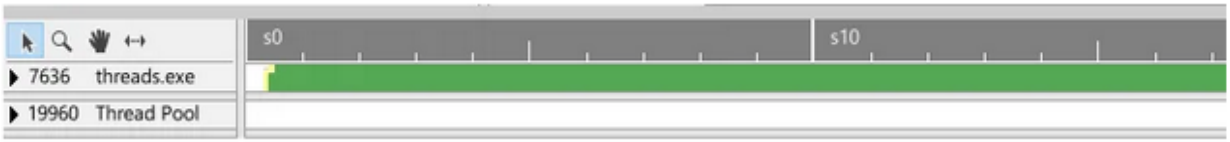
C++ 复制代码

```
void RunSequential()
{
    std::queue<CalcPiJob*> jobQ = GetJobsQ();
    while (!jobQ.empty())
    {
        CalcPiJob* job = jobQ.front();
        jobQ.pop();

        job->DoWork();
        delete job;
    }
}
```

我在 i7 4770K 上运行所有测试，它有 4 个内核和 8 个线程。所有时间均取自发布版本，以及来自调试版本的所有配置文件图像（用于说明工作量目的）。

顺序运行时间：*20692*毫秒




# 第一个工作线程

让有趣的部分开始。作为迈向多线程应用程序的一个简单步骤，我将只创建一个线程，与主线程共享工作负载。

这已经带来了一些需要注意的新概念，例如跨多个线程共享数据。我们使用std::mutex保护我们的数据访问，并使用std::scoped_lock锁定它（在 C++17 中引入。如果您的编译器不支持，请使用类似的std::lock_guard）。

您首先需要一些包含。

C++ 复制代码

```
// you should already have these.
#include <vector>
#include <queue>
```

```
#include <thread> // thread support
#include <mutex>  // mutex support
#include <atomic> // atomic variables
#include <future> // later on for std::async
```

C++ 复制代码

```
CalcPiJob* GetAndPopJob(std::queue<CalcPiJob*>& jobQ, std::mutex& mutex)
{
    std::scoped_lock<std::mutex> lock(mutex);
    if (!jobQ.empty())
    {
        CalcPiJob* job = jobQ.front();
        jobQ.pop();

        return job;
    }
    return nullptr;
}
```

GetAndPopJob完全按照它所说的去做，如果存在，它将获得一份工作并将其从队列中弹出。empty(),front() 并pop()在此方法中使用std::scoped_lock.

C++ 复制代码

```
void ExecuteJobsQ(std::atomic<bool>& hasWork,
    std::queue<CalcPiJob*>& jobQ,
    std::mutex& mutex)
{
    while (hasWork)
    {
        CalcPiJob* currentJob = GetAndPopJob(jobQ, mutex);
        if (currentJob)
        {
            currentJob->DoWork();
            delete currentJob;
        }
        else
        {
            hasWork = false;
        }
    }
}
```

ExecuteJobsQ将在主线程和工作线程中运行。它得到一个工作，执行它，并继续直到没有更多的工作要做。

C++ 复制代码

```
// global mutex for read/write access to Job Queue
static std::mutex g_mutexJobQ;

void RunOneThread()
{
    std::queue<CalcPiJob*> jobQ = GetJobsQ();

    std::atomic<bool> jobsPending = true;

    // Starting new thread
    std::thread t([&]() {
        ExecuteJobsQ(jobsPending, jobQ, g_mutexJobQ);
    });

    // main thread, also does the same.
    ExecuteJobsQ(jobsPending, jobQ, g_mutexJobQ);

    t.join();
}
```

一个工作线程运行时间：10396 毫秒

上图显示了作业的执行情况，首先是较大的作业，然后是中型作业，最后是较小的作业。这是将任务添加到队列中的顺序。

# 更多工作线程

现在这很好，所以让我们添加更多线程！多少？好吧，我知道我的 CPU 有 8 个线程，但是没有什么能保证它们只会为我的程序运行。操作系统时间片跨多个内核/线程执行程序，因此即使您创建的线程数超过最大 CPU 线程数，也不会有“问题”，因为操作系统会自行为它们切换执行时间。

C ++为我们提供确定有多少个并发线程我们的系统支持的一种方式，所以让我们只需要使用：
`std::thread::hardware_concurrency()`。

C++ 复制代码

```
void RunThreaded()
{
    // -1 to make space for main thread
    int nThreads = std::thread::hardware_concurrency() - 1;
    std::vector<std::thread> threads;

    std::queue<CalcPiJob*> jobQ = GetJobsQ();

    std::atomic<bool> hasJobsLeft = true;
    for (int i = 0; i < nThreads; ++i)
    {
        std::thread t([&]() {
            ExecuteJobsQ(hasJobsLeft, jobQ, g_mutexJobQ);
        });
        threads.push_back(std::move(t));
    }

    // main thread
    ExecuteJobsQ(hasJobsLeft, jobQ, g_mutexJobQ);

    for (int i = 0; i < nThreads; ++i)
    {
        threads[i].join();
    }
}
```

8 个线程的运行时间：2625 毫秒。

***8* 线程**

现在这是一个更好的视图。7 个工作线程与主线程一起处理所有作业。同样，我们首先看到较大的作业，然后是中型作业，然后是较小的作业。这是按照添加顺序进行处理的。

# 异步任务

使用std::async 生成任务时，我们不会手动创建线程，它们是从线程池生成的。

C++ 复制代码

```
void RunJobsOnAsync()
{
    std::vector<CalcPiJob*> jobs = GetJobVector();

    std::vector<std::future<void>> futures;
    for (int i = 0; i < jobs.size(); ++i)
    {
        auto j = std::async([&jobs, i]() {
            jobs[i]->DoWork();
            });
        futures.push_back(std::move(j));
    }

    // Wait for Jobs to finish, .get() is a blocking operation.
    for (int i = 0; i < futures.size(); ++i)
    {
        futures[i].get();
    }

    for (int i = 0; i < jobs.size(); ++i)
    {
        delete jobs[i];
    }
}
```

运行时间：2220 毫秒

# 概述

此时间表仅用作此特定情况的概览。当然，在实际应用中，结果各不相同。

| **测试运行** | **时间** (毫秒) | **改进** |
|---|---|---|
| 顺序的 | *20692* | 1.x |
| 一根线 | 10396 | 1.99 倍 |
| 螺纹 | 2625 | 7.88 倍 |
| 异步任务 | 2220 | 9.3 倍 |

示例代码是我对这个特定案例的探索，绝不是没有错误。但是看看代码如何跨多个线程运行，如何同步并充分利用我的系统是很有趣的。

所有屏幕截图都是使用程序的调试版本截取的，因此我们可以清楚地看到分析器中的工作量。为此，我使用了Superluminal Profiler。我发现它是一个了不起的轻量级分析器。您还可以免费使用英特尔的 VTune。

**从 GitHub 下载代码**

# 继续阅读

- 第 2 部分：探索 C++ 中的多线程续。

- 第 3 部分：探索 C++ 中的多线程：加载纹理
- 第 4 部分：探索 C++ 中的多线程：并行化光线追踪

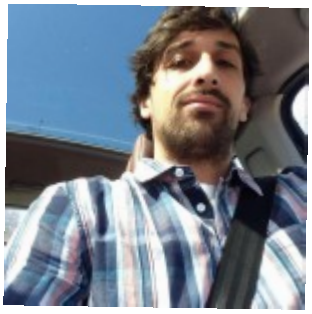
本文最初发表于http://mikeadev.net/2019/10/exploring-multi-threading-in-c

# 执照



# 分享

# 关于作者



**迈克尔·阿代索**
软件开发人员
英国
手表该会员
http://mikeadev.net/about-me/

# 评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

**非常好，但是...**
**Member 14660942** **19-Nov-19 21:39**

回复：非常好，但是...
**Michael Adaixo** 20-Nov-19 6:22

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。