79 How to make C++ run FASTER (with std-async)

本节会深入到一个项目中讨论如何通过多线程来提高性能。（可复习62 C++线程）

现代的硬件是能够并行处理的，这是为它们而设计的。如果你检查你使用的任何硬件，不管是手机电脑或者类似的东西，你会发现它们有不止一个CPU核心。这意味着你可以在同一时间并行执行指令，而不需要等待上一条指令完成后再执行下一条CPU指令，你可以在同一时间把事情做完，这整个的程序调度、多线程、并行化的过程是一个非常庞大的主题。

来看看如何通过充分利用我们的多核架构将一个现实世界开发的游戏引擎加载速度提高10倍。

1. 多线程

随着C++11的引入，我们在标准库中得到了许多有用的东西，这些东西可以帮助我们实现程序的多线程化，特别是我们将要讨论并了解的一个叫`std::asnyc`的东西，以及`futures`，它们是相互关联的。

做并行运行最难的是要找出彼此的依赖关系，并想清楚在不同的线程中放什么。因为你不能拿起来程序就说，每个程序都会在不同的线程上调用，一切都会很好——这并不是它真正的工作方式，在你的程序中，有一些东西能够被放在一个不同的线程中，这叫做 *worker thread*（**工作线程**），它可以独立地完成，而它什么时候开始、什么时候结束并不重要。但当它完成时，你可以得到一些反馈或者一些加载的新资源。

下面给出的例子中，你不需要对游戏引擎很熟悉，因为这适用于任何事务，但是游戏引擎是一个很好的游戏案例，因为它必须处理大量数据，而处理大量数据时，你需要去考虑策略或方法来减轻对CPU处理时间的影响。因为如果你能更好地控制所有的数据处理，你就能更聪明地使用它，你就能把它转换成线程处理，转换成你的CPU核心处理，把你的计算机作为一个整体来使用。（如果需要的话，你甚至可以使用你的GPU或类似的东西进行大规模的并行处理）

你做的越多，一切就会越快，这意味着你可以处理更多数据。一个现实中很好的例子就是：很多游戏有很长的加载时间， 因为有很多游戏资源要载入。而载入的过程不只是从光盘上读取文件，还可能涉及到解压缩，还可能涉及到把它发送到GPU，还可能涉及到在特定的环境下发生任何类型的转换。但其实每一个资源、纹理模型通常是**独立于彼此**的，这意味着它们都是很好的多线程候选场景。

每次我们加载模型的时候，就不需要等待上一个模型加载完成，再一个接一个地加载了，我们可以把它发送到一堆工作线程，并让它 *asynchronously* （异步)地发生。 这里的美妙之处在于你可以将所有的数据流化，你可以开始渲染你的关卡和世界的同时，在后台加载内容，有些内容甚至还没有完成加载，但玩家已经能够进入游戏世界行走。

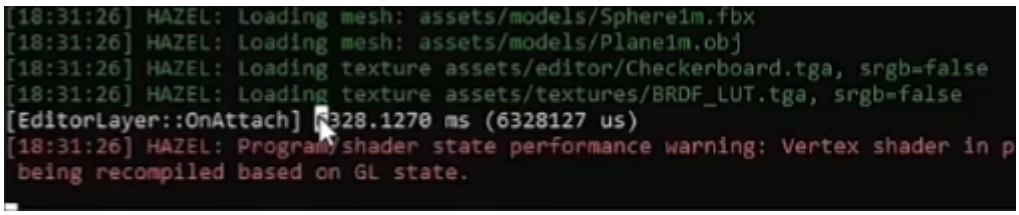
2. 如何用std::async来分割

Cherno用自己开发的Hazel引擎示范，其中有一个LoadMeshes函数是用于导入模型的，负责整个程序初始化的OnAttach函数启动后，会调用模型导入函数LoadMeshes，这个函数完成我们就可以开始渲染了。

我们对实际的应用程序启动，到在屏幕上渲染帧需要多长时间。于是这里使用了一个Timer（见63 Timing in C++）来测量这个OnAttach函数的时间：

```cpp
void EditorLayer::LoadMeshes()
{
    std::ifstream stream("src/Models.txt");
    std::string line;
    std::vector<std::string> meshFilepaths;
    while (std::getline(stream, line))
        meshFilepaths.push_back(line);

    for (const auto& file : meshFilepaths)
        m_Meshes.push_back(Mesh::Load(file));

}

void EditorLayer::OnAttach()
{
    Timer funcTimer("EditorLayer::OnAttach");

    m_Mesh.reset(new Mesh("assets/models/ZIL-130.fbx"));
    m_MeshMaterial.reset(new MaterialInstance(m_Mesh->GetMaterial()));

    LoadMeshes();
```

启动后，可以看到用时6.3s左右：



这是相当长的一段时间，可以通过使用`std::async`来做得更好。

如果我们回去看看需要做什么来实现多线程，感兴趣的地方应该是这部分for循环：

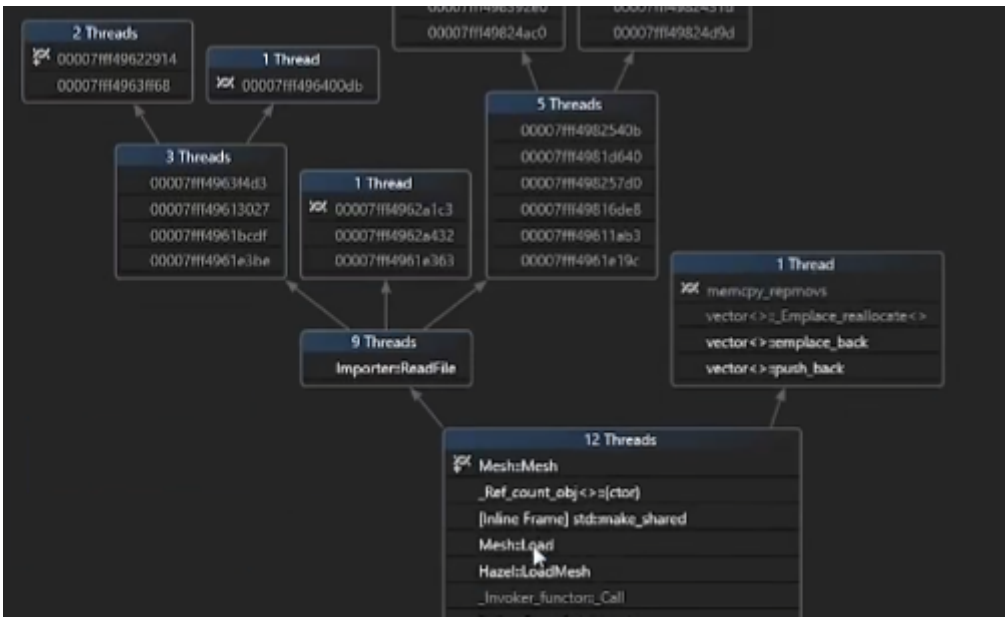
```cpp
void EditorLayer::LoadMeshes()
{
    std::ifstream stream("src/Models.txt");
    std::string line;
    std::vector<std::string> meshFilepaths;
    while (std::getline(stream, line))
        meshFilepaths.push_back(line);

    for (const auto& file : meshFilepaths)
        m_Meshes.push_back(Mesh::Load(file));
```

> 它会遍历每一个mesh的文件路径，它所做的只是加载网格，显然是非常独立的，并不依赖于之前加载的网格或类似的东西。

所以我们可以把它变成一个并行的for循环：

```C++
#include <future>

    for(const auto& file : meshFilepaths)
    {
        m_Future.push_back(std::async(std::launch::async, LoadMesh, m_Meshes,file));
    }
// 第一个参数：这是什么类型的job,设置启动类型为async很重要;
// 第二个参数：某种实际异步运行的函数
```

当你尝试多线程的时候，某些资源不能被同时访问，你必须锁定资源来做你想要做的事情，比如修改它然后解锁它，这样另一个线程才能访问它。方法是使用 *mutex* （互斥量），后面会提到。



优化后启动快了十倍。

可以看到，这种多线程对你的程序来说是非常有意的，可以充分利用你的硬件来提高速度。很多性能方面的优化都是关于充分利用你正在使用的硬件的，了解你在什么平台上发布代码，了解你的程序将在什么硬件上运行，然后利用这些优势。我们要处理的都是多核机器，利用这些线程可以让你的程序不只是顺序地执行一条条指令，而是把一些东西，推迟或分派到不同的线程，让计算机能够更快地处理这些东西。