**4.2 CPU与GPU间的交互**

**1，命令队列**

**命令队列**：每个GPU都至少维护着一个命令队列。CPU可利用命令列表将命令提交到这个队列中去。

命令队列被抽象为ID3D12CommandQueue接口表示。

通过填写D3D12\_COMMAND\_QUEUE\_DESC结构体来描述队列，调用ID3D12Device::CreateCommandQueue方法创建队列。

ComPtr<ID3D12CommandQueue> mCommandQueue;

D3D12\_COMMAND\_QUEUE\_DESC queueDesc = {}

queueDesc.Type = D3D12\_COMMAND\_LIST\_TYPE\_DIRECT;

queueDesc.Flags = D3D12\_COMMAND\_QUEUE\_FLAG\_NONE;

ThrowIfFailed(md3dDevice->CreateCommandQueue(&queueDesc, IID\_PPV\_ARGS(&mCommandQueue)))

IID\_PPV\_ARGS辅助宏定义如下：

#define IID\_PPV\_ARGS(ppType) \_\_uuidof(\*\*(ppType)), IID\_PPV\_ARGS\_Helper(ppType)

其中\_\_uuidof(\*\*(ppType))将获取(\*\*(ppType))的COM接口ID（全局唯一标识符，GUID）。

IID\_PPV\_ARGS辅助函数的本质是将ppType强制转换为void\*\*类型。

**2，命令列表**

利用ExecuteCommandLists可将命令列表里的命令添加到命令队列之中：

void ID3D12CommandQueue::ExecuteCommandLists(UINT count, ID3D12CommandList \* const \*ppCommandLists);

ID3D12GraphicsCommandList接口封装了一系列图形渲染命令，它实际上继承于ID3D12CommandList接口。

向命令列表中添加设置视口，清除渲染目标视图和发起绘制调用的命令：

mCommandList->RSSetViewports(1,&mScreenViewPort);

mCommandList->ClearRenderTargetView(mBackBufferView, Colors::LightSteelBlue, 0, nullptr)

mCommandList->DrawIndexedInstanced(36, 1, 0, 0, 0)

调用ExecuteCommandLists方法才会将命令真正地送入命令队列，供GPU在合适的时机处理。当命令都被加入命令列表之后，我们必须调用ID3D12GraphicsCommandList::Close方法来结束命令的记录。在调用ID3D12CommandQueue::ExecuteCommandLists方法提交命令列表之前，一定要将其关闭。

**3，命令分配器**

记录在命令列表内的命令，实际上是存储在与之关联的命令分配器ID3D12CommandAllocator上。当通过ID3D12CommandQueue::ExecuteCommandLists方法执行命令列表的时候，命令队列就会应用分配器里的命令。

命令分配器则由ID3D12Device接口来创建：

HRESULT ID3D12Device::CreateCommandAllocator(

D3D12\_COMMAND\_LIST\_TYPE type, REFIID riid, void \*\*ppCommandAllocator);

命令列表同样由ID3D12Device接口创建：

HRESULT ID3D12Device::CreateCommandList(UINT nodeMask,

D3D12\_COMMAND\_LIST\_TYPE type, ID3D12CommandAllocator \*pCommandAllocator,

ID3D12PipelineState \*pInitialState, REFIID riid, void \*\*ppCommandList);

我们可以创建出多个关联于同一命令分配器的命令列表，但是不能同时用它们来记录命令。当其中一个命令列表在记录命令时，必须关闭同一命令分配器的其他命令列表。

在调用ID3D12Device::CreateCommandList方法之后，我们就可以通过ID3D12GraphicsCommandList::Reset方法，安全地复用命令列表C占用的相关底层内存来记录新的命令集。

重置命令列表并不会影响命令队列中的命令，因为相关的命令分配器仍在维护着其内存中被命令队列所引用的系列命令。

向GPU提交了一整帧的渲染命令后，我们可能还要为了绘制下一帧而复用命令分配器中的内存。ID3D12CommandAllocator::Reset方法由此应运而生。在没有确定GPU执行完命令分配器中的所有命令之前，千万不要重置命令分配器。

**4，CPU与GPU间的同步**

CPU更改了资源R的位置信息，等待GPU使用，但是在GPU使用之前，CPU再次改变了R的位置信息，会造成一个严重的错误。解决此问题的一种方法是：强制CPU等待，直到GPU完成所有命令的处理，达到某个指定的围栏点为止。

我们将这种方法称为刷新命令队列，可以通过围栏来实现这一点。围栏用ID3D12Fence接口来表示，此技术能用于实现GPU和CPU间的同步。

HRESULT ID3D12Device::CreateFence(

UINT64 InitialValue, D3D12\_FENCE\_FLAGS flags,REFIID riid, void \*\*ppFence);

每个围栏对象都维护着一个UINT64类型的值，此为用来标记围栏点的整数。我们将此值设为0，每当需要标记一个新的围栏点时就将它加1。

UINT64 mCurrentFence = 0;

void D3Dapp::FlushCommandQueue(){

mCurrentFence++;

ThrowIfFailed(mCommandQueue->Signal(mFence.Get(), mCurrentFence));

If(mFence->GetCompleteValue() < mCurrentFence){

HANDLE eventHandle =

CreateEventEx(nullptr, false, false, EVENT\_ALL\_ACCESS);

ThrowIfFailed(mFence->SetEventOnCompletion(mCurrentFence, eventHandle));

WaitForSingleObject(eventHandle, INFINITE);

CloseHandle(eventHandle);

}

}

**5，资源转换**

当GPU的写操作还没有完成抑或甚至还没有开始，却开始读取资源，便会导致资源冒险。为此，Direct3D专门针对资源设计了一组相关状态。对某个资源执行写操作时，需要将它的状态转换为渲染目标状态；而要对资源进行读操作时，再把它的状态变为着色器资源状态。

通过命令列表设置转换资源屏障数组，即可指定资源的转换。资源屏障用D3D12\_RESOURCE\_BARRIER结构体表示。

CD3DX12\_RESOURCE\_BARRIER继承自D3D12\_RESOURCE\_BARRIER结构体，并添加了一些辅助方法，定义在d3d12x.h头文件中，这个文件不属于SDK核心部分，需要自己下载。

mCommandList->ResourceBarrier(1,

&CD3CX12\_RESOURCE\_BARRIER::Transition(CurrentBackBuffer(),D3D12\_RESOURCE\_STATE\_PRESENT, D3D12\_RESOURCE\_STATE\_RENDER\_TARGET))

**6，命令与多线程**

对于内含许多物体的庞大场景而言，仅通过一个构建命令列表来绘制整个场景会占用不少的CPU时间。

因此可以采用一种并行创建命令列表的思路。例如，我们可以创建4条线程，每条分别负责构建一个命令列表来绘制25%场景物体。

以下是一些在多线程环境中使用命令列表要注意的问题：

1，命令列表并非自由线程对象。也就是说，多线程既不能同时共享相同的命令列表，也不能同时调用同一命令列表的方法。所以，每个线程通常都只使用各自的命令列表。

2，命令分配器也不是线程自由的。每个线程都仅使用属于自己的命令分配器。

3，命令队列是线程自由的。多个线程可以同时访问同一命令队列，也能够同时调用它的方法。特别是每个线程都能同时向命令队列提交它们自己所生成的命令列表。

4，出于性能原因，应用程序必须在初始化期间，支出用于并行记录命令的命令列表最大数量。