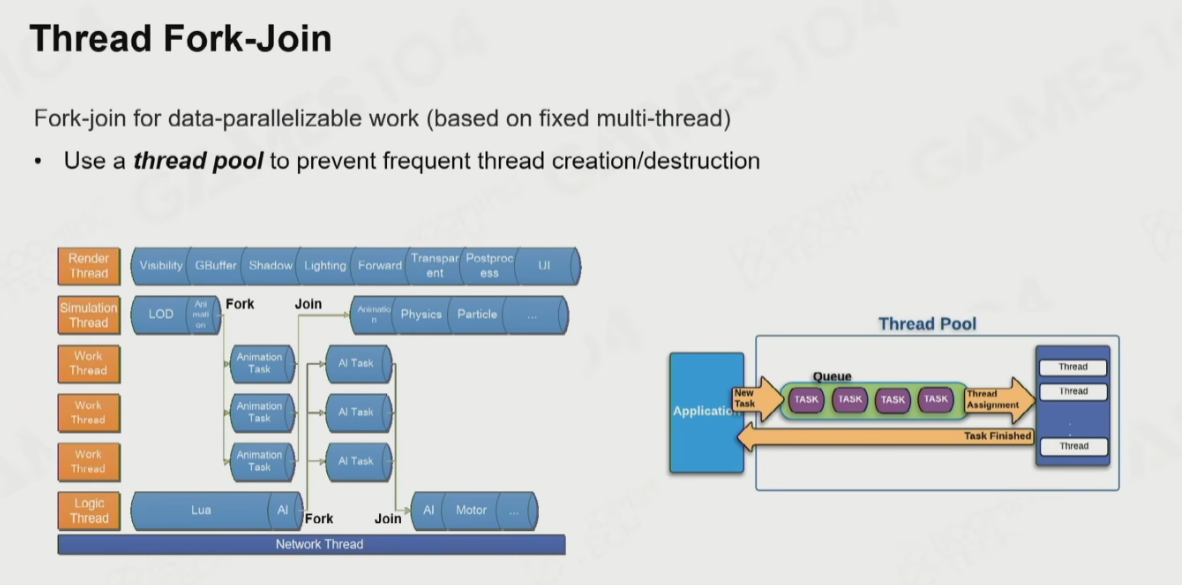
# **什么是JobSystem[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E4%BB%80%E4%B9%88%E6%98%AFjobsystem)**

## **并行编程[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E5%B9%B6%E8%A1%8C%E7%BC%96%E7%A8%8B)**

在游戏开发过程中我们经常会遇到要处理大量数据计算的需求，因此为了充分发挥硬件的多核性能，我们会需要用到并行编程，多线程编程也是并行编程的一种。

线程是在进程内的，是共享进程内存的执行流，线程上下文切换的开销是相当高的，大概有2000的CPU Circle，同时会导致缓存失效，导致万级别的CPU Circle，Job System的设计使用了线程池，一开始先将大量的计算任务分配下去尽量减少线程的执行流被打断，也降低了一些thread的切换开销。  
[](https://img2023.cnblogs.com/blog/1728741/202303/1728741-20230321183259771-207200700.png)

Unreal Unity大部分都是这种模型，分配了一些work thread 然后其他的线程往这个线程塞Task，相比fixed thread模式性能好一些，多出了Task的概念，Unity里称这个为Job。

## **Unity JobSystem[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "unity-jobsystem)**

通常Unity在一个线程上执行代码，该线程默认在程序开始时运行，称为**主线程**。我们在主线程使用JobSystem的API，去给worker线程下发任务，就是使用**多线程**。

通常Unity JobSystem会和Burst编译器一起使用，Burst会把IL变成使用LLVM优化的CPU代码，执行效率可以说大幅提升，但是使用Burst时候debug会变得困难，会缺少一些报错的堆栈，此时关闭burst可以看到一些堆栈，更方便debug。  
虽然并行编程有着种种的技巧，比如，线程之间沟通交流数据有需要加锁、原子操作等等的数据交换等操作。但是Unity为了让我们更容易的编写多线程代码，

通过一些规则的制定，规避了一些复杂行为，同时也限制了一些功能，必要时这些功能也可以通过添加attribute、或者使用指针的方式来打破一些规则。  
规定包括但不限于：

* 不允许访问**静态变量**
* 不允许在Job里调度子Job
* 只能向Job里传递**值类型**，并且是通过拷贝的方式从主线程将数据传输进Job，当Job运行结束数据会拷贝回主线程，我们可以在主线程的job对象访问Job的执行结果。
* 不允许在Native容器里添加托管类型
* 不允许使用指针
* 不允许多个Job同时写入同一个地方
* 不允许在Job里分配额外内存

## **应用场景[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF)**

基本上所有需要处理数据计算的场景都可以使用，我们可以用它做大量的游戏逻辑的计算，  
我们也可以用它来做一些编辑器下的工具，可以达到加速的效果。

# **细节[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E7%BB%86%E8%8A%82)**

## **接口[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E6%8E%A5%E5%8F%A3)**

unity官方提供了一系列的接口，写一个Struct实现接口便可以执行多线程代码，提供的接口包括：

* IJob：一个线程
* **IJobParallelFor**：多线程，使用时传入一个数组，根据数组长度会划分出任务数量，每个任务的索引就是数组元素的索引
* **IJobParallelForTransform**：并行访问Transform组件的，这是unity自己实现的比较特殊的读写Transform信息的Job，实测下来用起来貌似worker还是一个在动，但是经过Burst编译后快不少。
* IJobFor：几乎没用

IJobParallelFor是最常用的，对数据源中的每一项都调用一次 Execute 方法。Execute 方法中有一个整数参数。该索引用于访问和操作作业实现中的数据源的单个元素。

## **容器[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E5%AE%B9%E5%99%A8)**

Job使用的数据都需要使用Unity提供的Native容器，我们在主线程将要计算的数据装进NativeContainer里然后再传进Job。  
主要会使用的容器就是NativeArray，其实就是一个原生的数组类型，其他的容器这里暂时不提  
这些容器还要指定分配器，分配器包括

* Allocator.Temp: 最快的配置。将其用于生命周期为一帧或更少的分配。从主线程传数据给Job时，不能使用Temp分配器。
* Allocator.TempJob: 分配比 慢Temp但比 快Persistent。在四帧的生命周期内使用它进行线程安全分配。
* Allocator.Persistent: 最慢的分配，但只要你需要它就可以持续，如果有必要，可以贯穿应用程序的整个生命周期。它是直接调用malloc. 较长的作业可以使用此 NativeContainer 分配类型。

容器在实现Job的Struct里可以打标记，包括ReadOnly、WriteOnly，一方面可以提升性能，另一方面有时候会有读写冲突的情况，此时应该尽量多标记ReadOnly，避免一些数据冲突。

## **创建 使用[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E5%88%9B%E5%BB%BA-%E4%BD%BF%E7%94%A8)**

官方文档已经说的很好。  
[https://docs.unity3d.com/Manual/JobSystemCreatingJobs.html](https://docs.unity3d.com/Manual/JobSystemCreatingJobs.html" \t "https://www.cnblogs.com/FlyingZiming/p/_blank)  
对于ParallelFor的Schedule多了一些参数，innerloopBatchCount这个参数可以留意一下，可以理解为一个线程次性拿走多少任务。

## **Job之间互相依赖[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "job%E4%B9%8B%E9%97%B4%E4%BA%92%E7%9B%B8%E4%BE%9D%E8%B5%96)**

[https://docs.unity3d.com/Manual/JobSystemJobDependencies.html](https://docs.unity3d.com/Manual/JobSystemJobDependencies.html" \t "https://www.cnblogs.com/FlyingZiming/p/_blank)

其实执行了一个Job之后，在主线再执行另一个Job也不会性能差很多，并且易于debug，可以断点查看多个阶段执行过程中Job的数据情况，但是追求完美还是可以把依赖填上。

## **性能测试比较[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E6%80%A7%E8%83%BD%E6%B5%8B%E8%AF%95%E6%AF%94%E8%BE%83)**

笔者曾经做过简单的使用Job和不用Job的对比，通过打上Unity Profiler的标记，可以方便的在图表里查看运行开销。

Profiler.BeginSample("Your Target Profiler Name");// your code

Profiler.EndSample();

### **IJob[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "ijob)**

using System.Collections;

using System.Collections.Generic;

using Unity.Collections;

using Unity.Jobs;

using UnityEngine;

using Unity.Burst;

[BurstCompile]

public class JobTest : MonoBehaviour

{

public bool useJob;

// Update is called once per frame

void Update()

{

float startTime = Time.realtimeSinceStartup;

if (useJob)

{

NativeArray<int> result = new NativeArray<int>(1, Allocator.TempJob);//four frame allocate

MyJobSystem0 job0 = new MyJobSystem0();

job0.a = 0;

job0.b = 1;

job0.result = result;

JobHandle handle = job0.Schedule();

handle.Complete();

result.Dispose();

Debug.Log(("Use Job:"+ (Time.realtimeSinceStartup - startTime) \* 1000f) + "ms");

}

else

{

var index = 0;

for(int i = 0; i < 1000000; i++)

{

index++;

}

Debug.Log(("Not Use Job:"+ (Time.realtimeSinceStartup - startTime) \* 1000f) + "ms");

}

}

}

[BurstCompile]

public struct MyJobSystem0 : IJob

{

public int a;

public int b;

public NativeArray<int> result;

public void Execute()

{

var index = 0;

for(int i = 0; i < 1000000; i++)

{

index++;

}

result[0] = a + b;

}

}

使用IJob执行一项复杂的工作，没有使用job跑了2-4ms，使用job也是跑了2-4 ms，但是使用了job+burst，这个for循环的速度就变得只有0.2-0.8 ms了，burst对此优化挺大的。

### **IJobParallelFor[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "ijobparallelfor)**

using System;

using System.Collections;

using System.Collections.Generic;

using Unity.Collections;

using Unity.Jobs;

using UnityEngine;

public class JobForTest : MonoBehaviour

{

public bool useJob;

public int dataCount;

private NativeArray<float> a;

private NativeArray<float> b;

private NativeArray<float> result;

private List<float> noJobA;

private List<float> noJobB;

private List<float> noJobResult;

// Update is called once per frame

private void Start()

{

a = new NativeArray<float>(dataCount, Allocator.Persistent);

b = new NativeArray<float>(dataCount, Allocator.Persistent);

result = new NativeArray<float>(dataCount, Allocator.Persistent);

noJobA = new List<float>();

noJobB = new List<float>();

noJobResult = new List<float>();

for (int i = 0; i < dataCount; ++i)

{

a[i] = 1.0f;

b[i] = 2.0f;

noJobA.Add(1.0f);

noJobB.Add(2.0f);

noJobResult.Add(0.0f);

}

}

void Update()

{

float startTime = Time.realtimeSinceStartup;

if (useJob)

{

MyParallelJob jobData = new MyParallelJob();

jobData.a = a;

jobData.b = b;

jobData.result = result;

// 调度作业，为结果数组中的每个索引执行一个 Execute 方法，且每个处理批次只处理一项

JobHandle handle = jobData.Schedule(result.Length, 1);

// 等待作业完成

handle.Complete();

Debug.Log(("Use Job:"+ (Time.realtimeSinceStartup - startTime) \* 1000f) + "ms");

}

else

{

for(int i = 0; i < dataCount; i++)

{

noJobA[i] = 1;

noJobB[i] = 2;

noJobResult[i] = noJobA[i]+noJobB[i];

}

Debug.Log(("Not Use Job:"+ (Time.realtimeSinceStartup - startTime) \* 1000f) + "ms");

}

}

private void OnDestroy()

{

// 释放数组分配的内存

a.Dispose();

b.Dispose();

result.Dispose();

}

}

// 将两个浮点值相加的作业

public struct MyParallelJob : IJobParallelFor

{

[ReadOnly]

public NativeArray<float> a;

[ReadOnly]

public NativeArray<float> b;

public NativeArray<float> result;

public void Execute(int i)

{

result[i] = a[i] + b[i];

}

}

普通for寻找两个list，遍历list元素然后相加，数据量10万，每一个批次这里是处理1个execute， 不开job 2.48ms，开job 1.34ms，job开了burst就0.28ms。

### **IJobParalForTransform[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "ijobparalfortransform)**

using Unity.Burst;using Unity.Collections;using Unity.Jobs;using Unity.Mathematics;using UnityEngine;using UnityEngine.Jobs;

public class TransformJobs : MonoBehaviour{

public bool useJob;

public int dataCount = 100;

//public int batchCount;

// 用于存储transform的NativeArray

private TransformAccessArray m\_TransformsAccessArray;

private NativeArray<Vector3> m\_Velocities;

private PositionUpdateJob m\_Job;

private JobHandle m\_PositionJobHandle;

private GameObject[] sphereGameObjects;

//[BurstCompile]

struct PositionUpdateJob : IJobParallelForTransform

{

// 给每个物体设置一个速度

[ReadOnly]

public NativeArray<Vector3> velocity;

public float deltaTime;

// 实现IJobParallelForTransform的结构体中Execute方法第二个参数可以获取到Transform

public void Execute(int i, TransformAccess transform)

{

transform.position += velocity[i] \* deltaTime;

}

}

void Start()

{

m\_Velocities = new NativeArray<Vector3>(dataCount, Allocator.Persistent);

// 用代码生成一个球体,作为复制的模板

var sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere);

// 关闭阴影

var renderer = sphere.GetComponent<MeshRenderer>();

renderer.shadowCastingMode = UnityEngine.Rendering.ShadowCastingMode.Off;

renderer.receiveShadows = false;

// 关闭碰撞体

var collider = sphere.GetComponent<Collider>();

collider.enabled = false;

// 保存transform的数组,用于生成transform的Native Array

var transforms = new Transform[dataCount];

sphereGameObjects = new GameObject[dataCount];

int row = (int)Mathf.Sqrt(dataCount);

// 生成1W个球

for (int i = 0; i < row; i++)

{

for (int j = 0; j < row; j++)

{

var go = GameObject.Instantiate(sphere);

go.transform.position = new Vector3(j, 0, i);

sphereGameObjects[i \* row + j] = go;

transforms[i\*row+j] = go.transform;

m\_Velocities[i\*row+j] = new Vector3(0.1f \* j, 0, 0.1f \* j);

}

}

m\_TransformsAccessArray = new TransformAccessArray(transforms);

}

void Update()

{

//float startTime = Time.realtimeSinceStartup;

if (useJob)

{

// 实例化一个job,传入数据

m\_Job = new PositionUpdateJob()

{

deltaTime = Time.deltaTime,

velocity = m\_Velocities,

};

// 调度job执行

m\_PositionJobHandle = m\_Job.Schedule(m\_TransformsAccessArray);

//Debug.Log(("Use Job:"+ (Time.realtimeSinceStartup - startTime) \* 1000f) + "ms");

}

else

{

for (int i = 0; i < dataCount; ++i)

{

sphereGameObjects[i].transform.position += m\_Velocities[i] \* Time.deltaTime;

}

//Debug.Log(("Not Use Job:"+ (Time.realtimeSinceStartup - startTime) \* 1000f) + "ms");

}

}

// 保证当前帧内Job执行完毕

private void LateUpdate()

{

m\_PositionJobHandle.Complete();

}

// OnDestroy中释放NativeArray的内存

private void OnDestroy()

{

m\_Velocities.Dispose();

m\_TransformsAccessArray.Dispose();

}}

100+vec3，不用job 0.02ms，用job +burst 0.02ms  
1600+vec3，不用job 0.31ms，用job 0.07ms +burst 0.04ms  
1万+vec3，不用job 2.23ms，用job 0.35ms + burst 0.12ms  
1万+float3，不用job 2.55ms，用job 0.4ms  
100万+float3，不用job 199ms ，用job 40ms + burst 31ms  
100万+vec3，不用job 189ms ，用job 35ms + burst 31ms

# **高级技巧[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E9%AB%98%E7%BA%A7%E6%8A%80%E5%B7%A7)**

## **使用特定的数学库中的实现[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E4%BD%BF%E7%94%A8%E7%89%B9%E5%AE%9A%E7%9A%84%E6%95%B0%E5%AD%A6%E5%BA%93%E4%B8%AD%E7%9A%84%E5%AE%9E%E7%8E%B0)**

unity特定的数学库中的数据类型可以获取simd优化，比如vector3就可以换成float3，但是缺少的数学库，就要自己解决了，所以我一般就vector3。

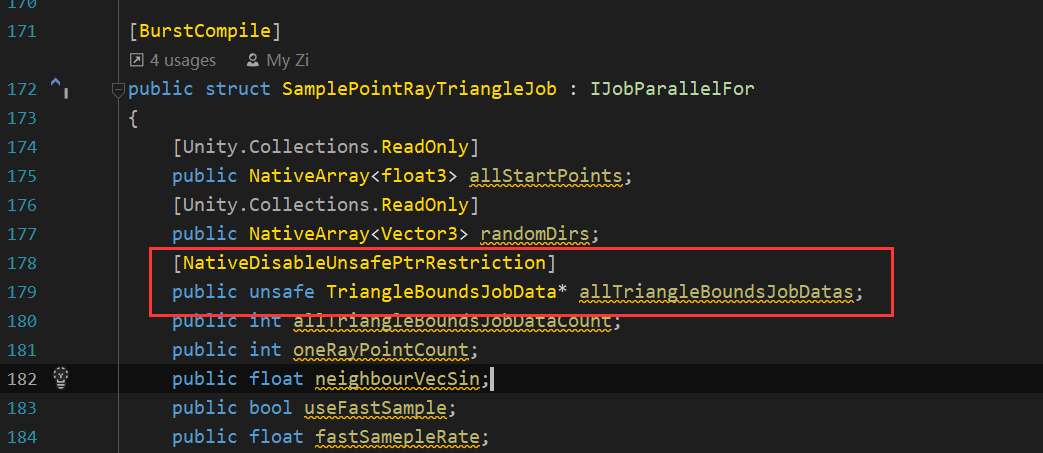
## **在合适的时机Schedule和Complete[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E5%9C%A8%E5%90%88%E9%80%82%E7%9A%84%E6%97%B6%E6%9C%BAschedule%E5%92%8Ccomplete)**

拥有作业所需的数据后就立即在作业上调用 Schedule，并仅在需要结果时才开始在作业上调用 Complete。最好是调度当前不与正在运行的任何其他作业竞争的、不需要等待的作业。例如，**如果在一帧结束和下一帧开始之间的一段时间没有作业正在运行，并且可以接受一帧延迟，则可以在一帧结束时调度作业，并在下一帧中使用其结果**。另一方面，如果游戏占满了与其他作业的转换期，但在帧中的其他位置存在大量未充分利用的时段，那么在这个时段调度作业会更加有效。

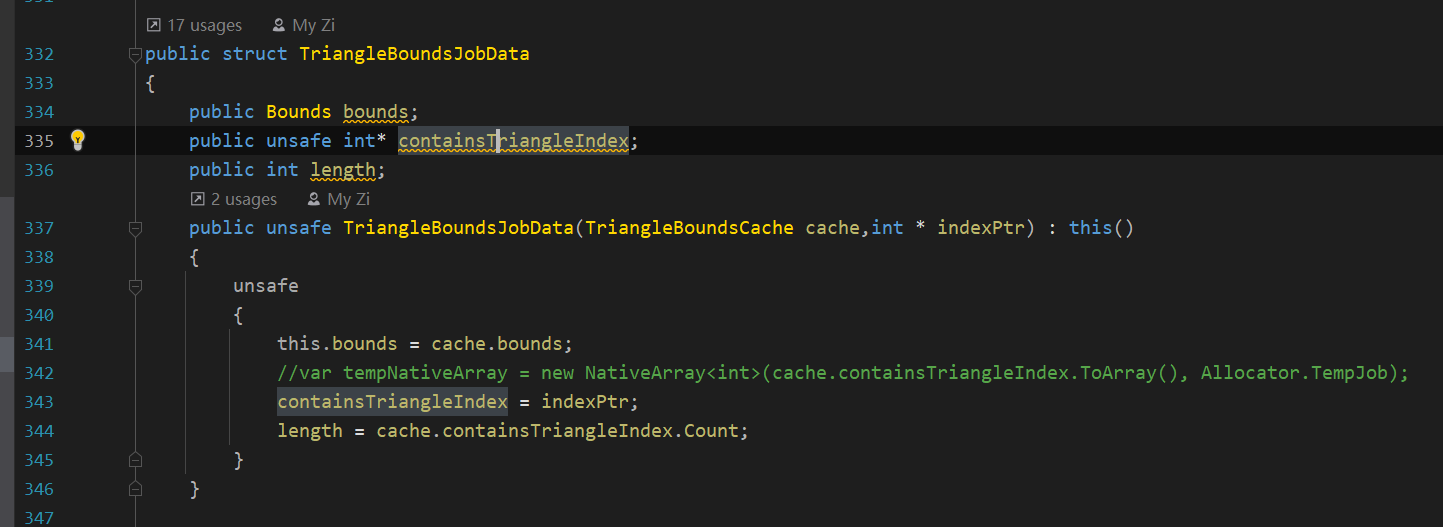
## **在单线程里运行JobSystem[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E5%9C%A8%E5%8D%95%E7%BA%BF%E7%A8%8B%E9%87%8C%E8%BF%90%E8%A1%8Cjobsystem)**

IJobParallelForExtensions可以调用Run方法，会将所有的Job放到一个Thread里执行，之前我们提到了Schedule的innerloopBatchCount参数，将它调到和数据源一样大，也是在一个Thread里执行，  
当我们的数据量小于1000，分配线程可能都觉得费劲，用单线程的JobSystem配合Burst效果可能更好。  
需要注意的是，如果我们出现了并行写入问题（多个Thread同时写一个位置），在单线程模式下是不会报错的。

## **使用NativeDisableUnsafePtrRestriction[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E4%BD%BF%E7%94%A8nativedisableunsafeptrrestriction)**

[](https://img2023.cnblogs.com/blog/1728741/202303/1728741-20230321183440090-1977380290.png)

打上这个标记后可以在Job里使用Unsafe代码块，使用指针  
有多个好处

* 可以不需要拷贝数组就把主线程的数据塞进子线程，对数据量大，需要频繁调用的可以考虑
* 可以包装一些托管内存，比如我这里就包装了一个二维数组，每个containsTriangleIndex其实是一个int的NativeArray  
  [](https://img2023.cnblogs.com/blog/1728741/202303/1728741-20230321183502665-1912258792.png)

如果struct里有NativeArray，这个struct放进NativeArray的时候会过不了安全检查。  
我这里是在主线程维护好了这些动态的数组，然后再传进了这个结构的。  
在unsafe代码块里，Native容器相关的API中有GetUnsafePtr可以获得指针。

SamplePointRayTriangleJob samplePointRayTriangleJob = new SamplePointRayTriangleJob();

samplePointRayTriangleJob.meshTriangles = jobMeshTriangles;

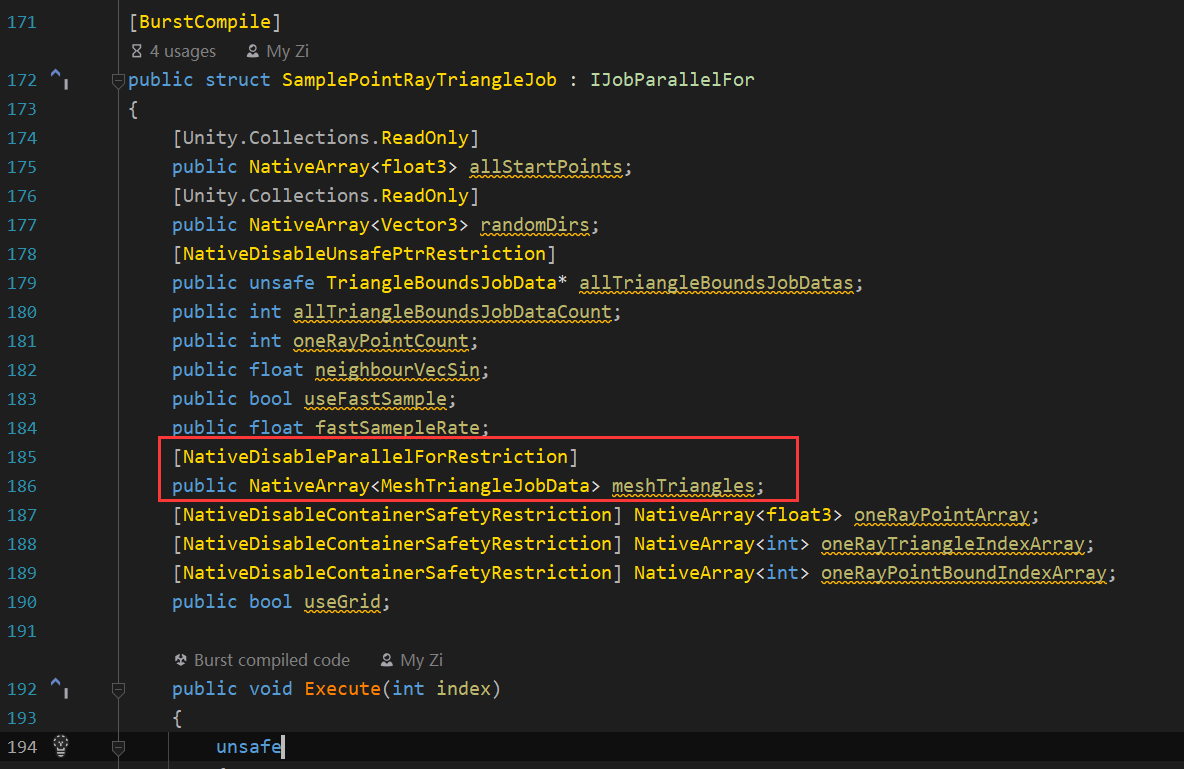
samplePointRayTriangleJob.randomDirs = jobRandomDirs;

samplePointRayTriangleJob.useGrid = useGrid;

samplePointRayTriangleJob.allStartPoints = startPoints;

samplePointRayTriangleJob.allTriangleBoundsJobDatas = (TriangleBoundsJobData\*)triangleBoundsJobDatas.GetUnsafePtr();

## **NativeDisableParallelForRestriction并行写入[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "nativedisableparallelforrestriction%E5%B9%B6%E8%A1%8C%E5%86%99%E5%85%A5)**

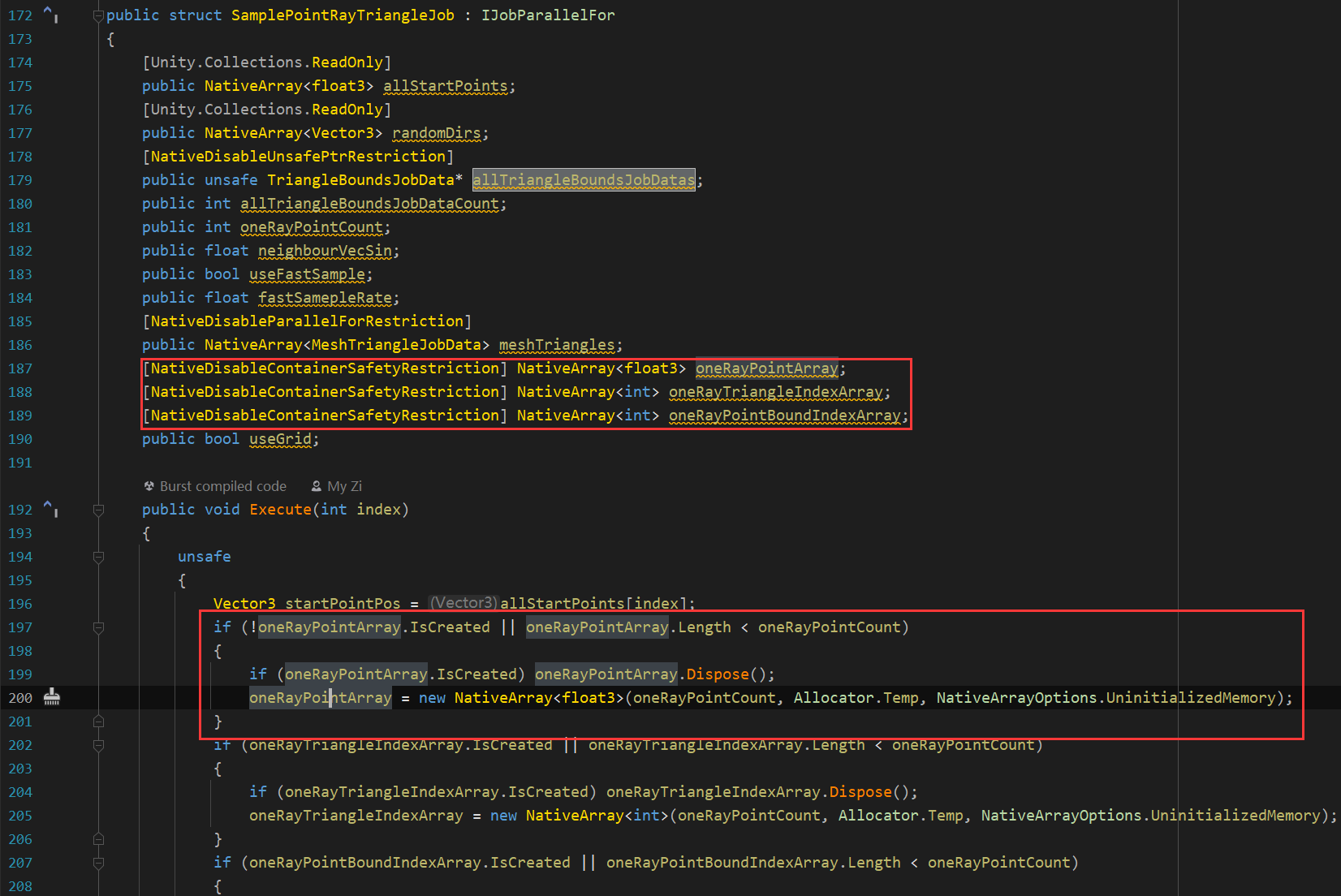
[](https://img2023.cnblogs.com/blog/1728741/202303/1728741-20230321183451684-2108836430.png)

打上这个标记后，多个Thread同时数组的同一个地方进行写入，unity不会阻拦，但是自己也要处理好逻辑问题。

举个例子：下面这篇文章里  
[https://blog.csdn.net/n5/article/details/123742777](https://blog.csdn.net/n5/article/details/123742777" \t "https://www.cnblogs.com/FlyingZiming/p/_blank)  
在Parallel Job里面进行光栅化三角形时，多个三角形有可能并行访问depth buffer/frame buffer的相同地方。这在多线程编程中属于race conditions，Job system内部会检测出来，会直接报错。

IndexOutOfRangeException: Index 219108 is out of restricted IJobParallelFor range [4392…4392] in ReadWriteBuffer.  
ReadWriteBuffers are restricted to only read & write the element at the job index. You can use double buffering strategies to avoid race conditions due to reading & writing in parallel to the same elements from a job.

## **NativeDisableContainerSafetyRestriction[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "nativedisablecontainersafetyrestriction)**

使用这个Attribute可以在子线程分配一块内存，比如我这里每个子线程是创建了一个数组来接受光线三角形求交，一根光线击中了多少个点，一个子任务会执行许多次光线遍历Mesh  
[](https://img2023.cnblogs.com/blog/1728741/202303/1728741-20230321183404964-181650427.png)

这个主要是博主在Github上学习Unity官方的MeshApiExample项目看到的案例，有点像StaticBatch  
可以查看这个链接：[把整个场景的Mesh合并](https://github.com/Unity-Technologies/MeshApiExamples/blob/master/Assets/CreateMeshFromAllSceneMeshes/CreateMeshFromWholeScene.cs" \t "https://www.cnblogs.com/FlyingZiming/p/_blank)

## **DeallocateOnJobCompletion[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "deallocateonjobcompletion)**

容器在job结束之后自动释放  
这个博主用的很少 基本都是主动释放  
可能在用非并行Job的时候 接受外面的NativeArray后自己不想管释放之类的。  
可以查看一个github上别人的案例看看：[案例](https://github.com/needle-mirror/com.unity.tiny.rendering/blob/a7a8674e8e19f56998a92c1c1c1a407b08086df4/Authoring/Unity.Tiny.Rendering.Authoring/LitMeshConversionJob.cs" \t "https://www.cnblogs.com/FlyingZiming/p/_blank)

## **自定义Native容器[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E8%87%AA%E5%AE%9A%E4%B9%89native%E5%AE%B9%E5%99%A8)**

[https://docs.unity3d.com/Manual/job-system-custom-nativecontainer-example.html](https://docs.unity3d.com/Manual/job-system-custom-nativecontainer-example.html" \t "https://www.cnblogs.com/FlyingZiming/p/_blank)

# **思考[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "%E6%80%9D%E8%80%83)**

## **JobSystem与ComputeShader相比 优势[#](https://www.cnblogs.com/FlyingZiming/p/17241013.html" \l "jobsystem%E4%B8%8Ecomputeshader%E7%9B%B8%E6%AF%94-%E4%BC%98%E5%8A%BF)**

JobSystem主要是利用CPU来降低计算负载，在数量级上远远比不上GPU，在前面的性能测试中数据到万以上就相当吃力了。  
ComputeShader是利用GPU来降低计算负载，，现在GPU Driven的技术也逐渐越来越多。

思考这两个的取舍主要应该看业务逻辑的数据流向，如果我们的数据是从CPU发起的，那么在把数据从CPU拷贝到GPU也是肯定是不如在CPU内做拷贝要快的，  
如果我们的计算的数据最后是给CPU做下步计算的，如果用GPU做计算就会出现CPU等GPU的回读问题，数据若停留在GPU，那么ComputeShader自然好。

另外就是考虑两个后端的硬件特性，CPU高主频，处理复杂的逻辑，大量的循环、分支判断上比GPU要有优势，数量级上则GPU更有优势。

最后也可以考虑一下易用性问题，如果用到了很多原本在CPU里的数学库，在JobSystem里都是可以直接用的，ComputeShader的话则需要自己实现一版，不过脚手架这种东西属于见仁见智，  
只要自己方便就好。