Entity Component System

概述

Entity Component System (ECS)是Unity面向数据技术栈(Data-Oriented Tech Stack)的核心。其主要分为三个部分：

- **Entity**: 实体(entities)，填充你游戏或程序的东西
- **Components** : 实体关联的数据(data)，其由数据自身组织(而不是由实体)。数据组织方式的不同是面向对象设计和面向数据设计的一个重要区别。
- **Systems**: 逻辑部分，把component data从当前状态(state)，转移到下一个状态，例如，一个系统可能通过速度乘以上一帧的时间来更新所有运动实体(moving entities)。

另外，本文也提了一点关于访问实体数据的方法, Accessing Entity Data和Jobs in ECS。

1 实体(Entities)

Entities是ECS架构的三个重要组成元素之一。它们表示你的游戏中单个的东西("things")。一个entity既不包含行为(behavior)也没有数据，它表示哪些数据块是属于一起的。**Systems**提供行为。**Components**存储数据。

一个Entity本质上是一个ID。你可以认为它是一个超轻量级的**GameObject**，甚至默认情况下没有名字。Enity IDs是稳定的(stable)。它们是仅有的存储别的Component或entity的引用(reference)的方法。

一个EntityManager管理一个World中的所有entities。一个EnityManager维护一个entities列表，并组织(organize)与entity关联的数据，以达到最优性能。

尽管一个entity没有类型(type)，但是entities的groups可以用与之关联的data components分类(categorized)。当你创建entities并向它们添加components时，EntityManager会跟踪存在于实体上的一个唯一(unique)的components组合。这样一个唯一的组合被称为*Archetype* (之后，可以在System中用它来过滤具有某些components的entities)。当你向一个entity添加components时，EntityManager会创建一个EntityArchetype结构体。你可以用一个已经存在的EntityArchetyp来创建新的符合这个archetype的entities。你也可以提前创建一个EntityArchetype，然后用它来创建entities。

1.1 Creating Entities 创建实体

创建一个entity最简单的方法是在Unity Editor中。你可以把GameObjects和Prefabs放置在场景中，在运行时(at runtime)把它们转换为entities。对于游戏的动态部分(dynamic parts)，你可以创建一个spawing 系统，在一个job中创建大量entities。最后，你可以使用EntityManager.CreateEntity函数，一次创建一个Entity。

1.1.1 Creating Entities with an EntityManager (使用EntityManager创建实体)

EntityManager.CreateEntity函数有很多重载版本，可以使用其中的任何一个创建一个entity。这个被创建的entity和创建它的EntityManagern在同一个World中(关于World见下文)。

你可以使用以下方式逐个地创建entities:

- 你可以使用ComponentType对象的数组来创建一个带有components的entity。
- 你可以使用EntityArchetype创建一个带有components的entity。
- 你可以使用Instantiate复制一个已经存在的entity，包括它的数据。
- 创建一个没有components的entity，然后给它添加components。

你也可以一次创建多个entities：

- 使用这个重载的CreateEntity: `public void CreateEntity(EntityArchetype archetype, NativeArray<Entity> entities)`，用新创建的具有相同archetype的entities来填充NativeArray。
- 通过Instantiate，使用一个已经存在的entity(包括其数据)来填充一个NativeArray。
- 利用CreateChunk，显示地创建用指定数量和给定archetype的entities填充的Chunks(相当于内存块)。

1.2 Adding and Removing Components (添加和移除Components)

一个entity被创建后，你可以添加或移除components。当你这样做时，受到影响的entities的archetype发生了改变，EntityManager必须把修改的数据(altered data)移到一个新的内存块(Chunk of memory)，并压缩原始内存块里的component 数组 (原文：as well as condense the component arrays in the original Chunks)。

一个entity上发生的引起结构改变的变化，即，会改变SharedComponentData并销毁这个entity的添加或移除components的操作，不可以在一个Job内完成(cannot be done inside a job)，这是因为这些操作会使Job正在工作的数据无效。相反，你可以添加命令(commands)来使这些类型的变化成为一个EntityCommandBuffer，并在Job完成后执行这个命令缓存(command buffer)。

EntityManager提供了从单个entity或所有在一个NativeArray中的entities中移除一个component的方法。详细见下文关于**Components**的章节。

1.3 Iterating entities 迭代实体

迭代遍历具有某些components的实体，是ECS架构中很重要的工作。详见下文关于**Assesing entity Data**部分。

1.4 World

一个`World`拥有一个EntityManager和一个ComponentSystems(见下文)。你想创建多少个`World`对象，你就可以创建多少个。通常你会创建一个simulation `World`和一个rendering `World`或presentation `World`。

默认情况下，当进入**Play Mode**(播放模式)时，我们会创建一个`World`，并用项目中所有可用的`ComponentSystem`对象来填充它。但是你可以禁用创建这个默认的`World`，并通过一个全局定义(via a global define)来用你自己的代码代替它。

- **Default World creation code** (默认`World`的创建代码，查看文件:*Packages/com.unity.entities/Unity.Entities.Hybrid/Injection/DefaultWorldInitialization.cs)*)
- **Automatic bootstrap entry point** (自动引导入口点，查看文件: *Packages/com.unity.entities/Unity.Entities.Hybrid/Injection/AutomaticWorldBootstrap.cs*)

2 Components (组件)

ECS中的Component，组件，和经典Unity中的组件(如: MonoBehavior)不同，因为这里的Component不含行为。

Components是ECS架构中的三种主要元素之一。它们表示你的游戏的数据(the data)。而上文说过的**Entities**本质上是ID，用来索引你的components。下文提到的**Systems**提供行为。

具体地说，在ECS中，一个component是一个实现了以下任意一种"标记接口"("marker interfaces")的结构体(struct):

- IComponentData
- ISharedComponentData
- ISystemStateComponentData
- ISharedSystemStateComponentData

Entity把你的entity对应的唯一的components纵使，组织进一个**Archetype**中。它把所有具有相同archetype的entities的components一起存在一个内存块中，叫作**Chunks**(数据块)。所以一个给定的Chunk中的entities有相同的component archetype。

Shared components (共享组件)是一种特殊的data component (数据组件)，你可以基于shared component中的具体值来细分(subdivide)entities。当你向一个entity添加一个shared component时，EntityManager把所有具有相同shared data **values**的entities放进同一个Chunk中。 Shared components允许你的systems一起处理这些相似的entities(原文: Shared components allow your systems to process like entities together) 。例如，共享组件`Rendering.RenderMesh`，属于Hybrid.rendering package，定义了一些字段(fields)，包括 **mesh, material, receiveShadows**等。当渲染时，它能够最大效率地一起处理所有 *这些字段具有相同值的* 三维物体。因此这些属性在一个shared component中指定了，EntityManager可以把所有匹配的entities在内存中放在一起，这样渲染系统就可以高效地遍历它们(entities)。

注意: 过度使用shared components可能导致较差的Chunk利用率，因为它涉及一个Chunks数量的组合膨胀(combinatorial expansion)，这些Chunks有的是根据archetype被要请分配的，有的是根据每一个shared component字段(field)的不同值(unique value)请求分配的。应该避免向shared component 添加不必要的字段。使用Entity Debugger查看当前Chunk的利用率。

如果你在一个entity上添加或移除component，或者修改SharedComponent的值，EntityManager会把这个entity移动到不同的Chunk中，必要时会创建新的Chunk。(这个过程可能会造成性能问题么)

System状态(state)Components表现得和普通的components或shared components一样，除了当你销毁entities时，EntityManager不会移除任何system state components，也不会回收entity ID，直到它们被移除。这点行为上的不同 允许一个system清理它内部的状态或者当一个entity被销毁时，释放资源。

2.1 General Purpose Components 普通用途的组件

2.1.1 ComponentData

Unity中的ComponentData (在标准ECS术语中也称为Component)是一个结构体(struct)，仅包含一个entity的实例数据(instance data)。ComponentData不能包含方法。这有点像经典Unity中的只包含变量的(only contains variables)组件。

Unity ECS 提供了一个接口`IComponentData`，你的Component需要实现这个接口。如上文所说，这个接口中仅起“标记作用”("Marker interface")。

2.1.2 IComponentData

经典的Unity组件(如，MonoBehaviour)是面向对象的类，里面既包含数据也包含方法(即行为)。`IComponentData`是一个纯ECS风格的组件(component)，意味着它只有数据，没有方法。`IComponentData`是一个结构体而不是一个类，意味着这是按值拷贝而不是引用拷贝。你需要用下面的方法来修改数据:

```
var transform = group.transform[index]; // Read

transform.heading = playerInput.move; // Modify
transform.position += deltaTime * playerInput.move *
settings.playerMoveSpeed;

group.transform[index] = transform; // Write (注意这里)
```

`IComponentData`结构体不能包含托管对象的引用。因为所有`ComponentData`存在于一个简单的没有垃圾回收器追踪的内存块里。

查看文件: */Packages/com.unity.entities/Unity.Entities/IComponentData.cs*。

2.2 Shared Components

2.2.1 Shared ComponentData

`IComponentData`适合在不同entities之间变化的数据，如存储一个`World`的位置。

而当许多entities有一些共性时，`ISharedComponentData`是有用的，例如在Boid demo中，从同一个Prefab中实例化了许多entities， 因此`RenderMesh`(一个SharedComponentData)在许多 `Boid` entities中是相同。

```
[System.Serializable]
public struct RenderMesh : ISharedComponentData
{
    public Mesh                 mesh;
    public Material             material;

    public ShadowCastingMode    castShadows;
    public bool                 receiveShadows;
}
```

`ISharedComponentData`最大的优点是每一个entity的内存成本几乎为0。

我们使用`ISharedComponentData`把所有使用相同`InstanceRenderer`数据的entities组织在一起，然后高效地提取出所有渲染矩阵。结果，代码简单、高效，因为数据布局和我们访问它们的方式一样。

- `RenderMeshSystemV2` (查看: *Packages/com.unity.entities/Unity.Rendering.Hybrid/RenderMeshSystemV2.cs*)

2.2.2 关于SharedComponentData的注意事项

- 具有相同`SharedComponentData`的entities放在同一块Chunk中。`SharedComponentData`的索引，每个`Chunk`只存储一次。因此对于`SharedComponentData`，每个entity的内存开销几乎为0.
- 使用`EntityQuery`，我们可以遍历具有相同类型的entities。
- 另外，使用`EntityQuery.SetFilter()`，我们可以遍历所有具有特定`SharedComponentData`值的entities。由于这些entities在内存中是放在一块的，遍历开销很小。
- 使用`EntityManager.GetAllUniqueSharedComponents`，我们可以获取一个活着的entity上的unique `SharedComponentData`。
- `SharedComponentData`是自动引用计数的。
- `SharedComponentData`应该很少改变。因为修改一个`SharedComponentData`会涉及到使用memcpy为entity拷贝所有`ComponentData`到不同的`Chunk`中。总之，就是会造成大量内存拷贝发生。

2.3 System state Components

2.3.1 SystemStateComponents

`SystemStateComponents`的目的是允许你跟踪system内部的资源，有机会合理地创建和销毁这些资源，而不需要依赖一个个的回调(callbacks)。

`SystemStateComponents`和`SystemStateSharedComponentData`分别与`ComponentData`、`SharedComponentData`很像， 除了在一个重要的方面:

1. 当一个entity被销毁时，`SystemStateComponentData`不会被删除。

一个`DestroyEntity`通常是这样一个工具，用于：

1. 找到所有引用这个给定的entity ID的components。
2. 删除这些components。
3. 回收利用entity ID。

然而，如果使用`SystemStateComponentData`，它不会被移除。这给system一个机会来清理任何和一个entity ID关联的资源或状态。这个entity ID直到所有的`SystemStateComponentData`被删除了，才会被回收利用。

2.3.2 Motivation (动机)

- Systems可能需要保持一个基于`ComponentData`的内部状态。
- 当别的Systems改变了值或状态时，Systems需要能够管理这个状态。
- 没有回回调 "No callbacks" 是ECS设计规则的一个重要元素。

2.3.3 Concept

`SystemStateComponentData`的一般用途是提供内部状态的情况下，反映(mirror)用户 component。

例如，给定：

- FooComponent (`ComponentData`，由用户赋值)
- FooStateComponent(`SystemComponentData`，由system赋值)

2.3.3.1 Detecting Component Add

当用记添加FooComponent时，FooStateComponent不存在。FooSystem `OnUpdate`函数查询没有FooStateComponent的FooComponent，并推断出这些FooComponents被添加了。在此时，FooSystem将添加FooStateComponent和任何需要的内部状态(internal state)。

2.3.3.2 Detecting Component Remove

当用户移除FooComponent时，FooStateComponent依然存在。FooSystem`OnUpdate`函数查询没有FooComponent的FooStateComponent，并推断出这些FooComponent已经被移除了。这时，FooSystem将移除FooStateComponent并修改任何必要的内部状态。

2.3.3.3 Detecting Destroy Entity

如上所述，`DestroyEntity`实际上是一个工具，用于：

1. 找到所有引用这个给定的entity ID的components。
2. 删除这些components。
3. 回收利用entity ID。

然而，`SystemStateComponentData`没有被移除，并且entity ID不会被回收，直到最后一个component被删除。这给了system机会以与普通component移除一样的方式来清除内部状态。

2.3.4 SystemStateComponent

一个`SystemStateComponent`和`ComponentData`类似，使用方法也类似:

```
struct FooStateComponent : ISystemStateComponentData
{
}
```

`SystemStateComponent`的可见性与一个普通的component的控制方法一样(`private`, `public`, `internal`)。然而，一般情况下，我们希望一个`SystemStateComponent`在创建它的system外部应该是`ReadOnly`。

2.3.5 SystemStateSharedComponent

一个`SystemStateSharedComponent`和`SharedComponentData`类似，使用方法也相似。

```
struct FooStateSharedComponent :
ISystemStateSharedComponentData
{
  public int Value;
}
```

2.4 Dynamic Buffer Components (动态缓存组件)

2.4.1 Dynamic Buffers

一个`DynamicBuffer`是一种component data类型，允许一个长度可变、可伸展的buffer，与一个entity相关联。它表现得像这样一个component type: 有一个内部容量(internal capacity)，可以承载某些数量的元素，但当内部容量耗尽时，又可以分配堆上的内存块(a heap memory block)。

当使用这种方法时，内存管理是全自动的。和`DynamicBuffers`相关联的内存由`EntityManager`管理，这样内一个`DynamicBuffer` component被移除时，减除的堆内存(heap memory)会被自动释放。

`DynamicBuffers`替代了(已删除的)固定数组。

2.4.2 Declaring Buffer Element Types

为了声明一个`Buffer`，你可以用将要放进此`Buffer`内的元素的类型(type)来声明它。如下：

```
// This describes the number of buffer elements that should be reserved
// in chunk data for each instance of a buffer. In this case, 8 integers
// will be reserved (32 bytes) along with the size of the buffer header
// (currently 16 bytes on 64-bit targets)
[InternalBufferCapacity(8)]
public struct MyBufferElement : IBufferElementData
{
    // These implicit conversions are optional, but can help reduce typing.
    public static implicit operator int(MyBufferElement e) { return e.Value; }
    public static implicit operator MyBufferElement(int e) { return new MyBufferElement { Value = e }; }

    // Actual value each buffer element will store.
    public int Value;
}
```

尽管描述元素的类型而不描述`Buffer`本身的类型看起来很奇怪，但是这种设计为ECS提供了两个重要的优势：

1. 支持拥有多于一个`float3`(或者别的类型)类型的`DynamicBuffer`。只要元素被顶层的结构体唯一地包装起来，你就可以添加任意数量的`Buffers`，且利用相同的值类型(value types，此处应该是指一个普通的类型)。
2. 可以在`EntityArchetype`中包含`Buffer`元素的类型，它通常表现得就像有了一个component一样。

2.4.3 Adding Buffer Types To Entitites

为一个entity添加一个buffer，你可以使用普通的为entity添加component的方法。
使用 **AddBuffer()**

```
entityManager.AddBuffer<MyBufferElement>(entity);
```

Using an archetype

```
Entity e = entityManager.CreateEntity(typeof(MyBufferElement));
```

2.4.4 Accessing Buffers (访问缓存)

有一些方法来访问`DynamicBuffers`：

Direct, main-thread only access (只能在主线程中进行的直接访问)

```
DynamicBuffer<MyBufferElement> buffer =
entityManager.GetBuffer<MyBufferElement>(entity);
```

Entity based access (基于Entity的访问)

你也可以从一个`JobComponentSystem`中访问每一个entity上的`Buffers`：

```
var lookup = GetBufferFromEntity<EcsIntElement>();
    var buffer = lookup[myEntity];
    buffer.Append(17);
    buffer.RemoveAt(0);
```

Reinterpreting Buffers (experimental) (重新解释Buffers，试验阶段)

`Buffers`可以被解释为相同大小的另一个类型的Buffer。目的是为了摆脱包装类型(wrapper element types)。为了重新解释一个buffer，只需调用`Reinterpret<T>`：

```
var intBuffer = entityManager.GetBuffer<EcsIntElement>
().Reinterpret<int>();
```

重新解释的缓冲区带有原始缓冲区(original `Buffer`)的安全句柄，可以安全地使用。

它们使用相同的底层`BufferHeader`，因此修改一个重新解释的`Buffer`将会立即反映正别的相关Buffer上。

注意，这个过程没有类型检查，因此把同一个缓存重新解释为一个`uint` buffer或`float` buffer·都是可能的。

3 System (系统)

一个**System**，也即ECS中的*S*，提供了逻辑，来把component data从当前状态转换到下一个状态。例如，一个system可能更新所有运动实体(moving entities)的位置：速度乘以距离上一帧的时间间隔。

3.1 Component Systems

Unity中的一个`ComponentSystem` (标准ECS术语中的system)完成实体上的操作。一个`ComponentSystem`不能包含实例数据(instance data)。它像是经典Unity中只包含有方法的组件。一个`ComponentSystem`负责更新所有匹配某个components集合(set)的entities。

Unity ECS提供了一个抽象类`ComponentSystem`，你的system要继承它。

查看：*/Packages/com.unity.entities/Unity.Entities/ComponentSystem.cs*.

3.2 Job Component Systems

3.2.1 Automatic job dependency management (自动的任务依赖管理)

管理依赖是很困难的。但`JobComponentSystem`自动完成了这些。简单规则：来自不同systems的jobs可以并行地读取来自相同类型的`IComponentData`的数据。但如果一个job要写入数据，则这些jobs就不能并行地运行了，而是根据它们之间的依赖进行调度。

```
public class RotationSpeedSystem : JobComponentSystem
{
    [BurstCompile]
    struct RotationSpeedRotation : IJobForEach<Rotation, RotationSpeed>
    {
        public float dt;

        public void Execute(ref Rotation rotation, [ReadOnly]ref RotationSpeed speed)
        {
            rotation.value = math.mul(math.normalize(rotation.value), quaternion.axisAngle(math.up(), speed.speed * dt));
        }
    }

    // Any previously scheduled jobs reading/writing from Rotation or writing to RotationSpeed
    // will automatically be included in the inputDeps dependency.
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        var job = new RotationSpeedRotation() { dt = Time.deltaTime };
        return job.Schedule(this, inputDeps);
    }
}
```

3.2.2 How does this work? (它是怎么工作的？)

所有的jobs和systems都会声明它们会从什么类型的ComponentTypes中读取数据，或向其写入数据。当一个JobComponentSystem返回一个JobHandle时，它会自动地被注册到`EntityManager`和所有的types(包括关于它是读取还是要写入)。

如果一个system向一个component `A`写入数据，而稍后另一个system要从component `A`中读取数据，则后一个`JobComponentSystem`会浏览它将要读取的类型的列表，并向你传递一个关于第一个system的job的依赖。

> *上段文字原文*: *Thus if a system writes to component A, and another system later on reads from component A, then the JobComponentSystem looks through the list of types it is reading from and thus passes you a dependency against the job from the first system.*

`JobComponentSystem`只是简单地在需要的地方把jobs链接(chain)起来，因此不会造成主线程的停顿。但是如果一个non-job `ComponentSystem`访问这些数据(the same data)时，会发生么？因为所有的访问(access)都被声明了，在调用`OnUpdate`之前，`ComponentSystem`自动地完成这个system使用的所有components相关的的jobs。

3.2.3 Dependency management is conservative & deterministic

依赖关系管理是保守的(conservative)。`ComponentSystem`仅仅追踪曾被使用过的`EntityQuery`对象，并基于它们来存储要写入或读取的类型。

当在单个system中调度多个jobs时，必须把依赖(dependencies)传递到所有jobs中，即使不同的jobs可能几乎没有依赖关系。如果因此而造成性能问题，最好的解决方案是把一个system分成两个。

依赖关系管理方法是保守的。在提供一个非常简单的API的同时，它承诺了具有确定性和正确的行为(deterministic and correct behaviour)。

3.2.4 Sync points (同步点)

所有结构性的改变(structural changes)都有严格的同步点(hard syn points)。`CreateEntity`, `Instantiate`, `Destroy`, `AddComponent`, `RemoveComponent`, `SetSharedComponentData`都有有严格的同步点。例如，这意味着所有由`JobComponentSystem`调度的jobs会成创建the entity前完成。这会自动发生。因此，在一帧的中间调用`EntityManager.CreateEntity`可能导致大小的停顿来等待之前在这个`World`中已调度的所有jobs完成。

查看EntityCommandBuffer(见下文)可以找到有关于在游戏运行时创建entities时避免同步点的方法。

3.2.5 Multiple Worlds (多个Worlds)

每一个`World`有它自己的`EntityManager`和一个独立的`JobHandle`依赖管理的集合。

一个World中的严格同步点不会影响别的`World`。因此，对于流式和程序生成的情形(for streaming and procedural generation scenarios)，可以在一个`World`中创建entities，然后再在一帧的开始处的一个事务中(in one transaction at the beginning of the frame)，把它们移到别一个`World`中，这是很有用的。

3.3 Entity Command Buffer (实体命令缓存)

`EntityCommandBuffer`类解决两个重要的问题：

1. 在一个job中，你不能访问`EntityManager`。
2. 当你访问`EntityManager`(比如，创建一个entity)时，会使所有注入的数组(injected arrays)和`EntityQuery`对象无效。

`EntityCommandBuffer`可以允许你把一些(来自job或main thread)的变化入队(即放入一个队列)，稍后在主线程上才实际完成/生效(take effect)。有两种方法来使用一个`EntityCommandBuffer`:

在主线程上更新的`ComponentSystem`的子类有一个可用的且自动调用的`PostUpdateCommands`。要使用它，仅需要引用此属性并把你的变化(changes)入队。当system的`Update`函数返回时，这些变化会被立即自动地应用到the World。

这有个例子：

```
PostUpdateCommands.CreateEntity(TwoStickBootstrap.BasicEnemyArchetype);
PostUpdateCommands.SetComponent(new Position2D { Value = spawnPosition });
PostUpdateCommands.SetComponent(new Heading2D { Value = new float2(0.0f, -1.0f) });
PostUpdateCommands.SetComponent(default(Enemy));
PostUpdateCommands.SetComponent(new Health { Value = TwoStickBootstrap.Settings.enemyInitialHealth });
PostUpdateCommands.SetComponent(new EnemyShootState { Cooldown = 0.5f });
PostUpdateCommands.SetComponent(new MoveSpeed { speed = TwoStickBootstrap.Settings.enemySpeed });
PostUpdateCommands.AddSharedComponent(TwoStickBootstrap.EnemyLook);
```

如你所见，这个API和`EntityManager` API很像似。在这种模式下，这种自动的(automatic)`EntityCommandBuffer`像是一个便利的工具，使你能够不必使system中的数组失效，同时能对the World作出变化(changes)。

对于jobs，你必须在主线程上从一个entity command buffer system中请求`EntityCommandBuffer`，然后把它传递给jobs。当`EntityCommandBufferSystem`更新时(即产生了变化)，command buffers将会在主线程中以它们被创建的顺序执行。需要这个额外的步骤，以便集中(centralized)内存管理(memory management)，并且生成的entities和components的确定性(determinism)才可能被保证。

3.4 System Update Order

可以使用Component System Groups来指定你的systems的更新顺序(update order)。你在一个system类的声明上放置属性`[UpdateInGroup]`，便把这个system放进一个group中了。然后，你可以使用`[UpdateBefore]`和`[UpdateAfter]`属性来指定组内的更新顺序。

ECS框架创建了一系列default system groups，你可以在一帧的正确阶段(correct phase)使用它们来更新你的systems。你可以把一个group `A`嵌套在另一个group `B`中，这样group `A`中的所有systems都将会在正确阶段按照group `A`在group `B`中的顺序来更新。

3.4.1 Component System Groups

`ComponentSystemGroup`类表示一个应该按照特定顺序一起更新的相关联的component systems列表。`ComponentSystemGroup`继承自`ComponentSystemBase`，因此，在许多重要方面，它表现得和一个component system很像：它可以相对于别的systems被排序，有一个`OnUpdate`方法等。更贴切地讲，这意味着一个component system groups可以被嵌套在别的component system groups中，形成一个层次结构。

3.4.2 System Ordering Attributes

有一些现有的system ordering attributes，它们的语义(semantics)和限制(restrictions)稍微有点不同。

- `[UpdateInGroup]`： 指定一个system应该属于的ComponentSystemGroup。如果这个属性被忽略了，这个system会被自动添加到default World的`SimulationSystemGroup`。
- `[UpdateBefore]`和`[UpdateAfter]`：指定systems相对于组(group)内别的systems的顺序。而跨组的两个systems之间的排序是由包含这两个systems的最深的组决定 的，例如：如果SystemA在GroupA中，SystemB在GroupB中，而GroupA和GroupB都在GroupC中，则GroupA和GroupB的排序隐含地决定了SystemA和SystemB的相对排序;不需要对这两个Systems进行显式排序。
- `[DisableAutoCreation]`：阻止这个system在default world初始化时被创建。你必须自己显式地创建和更新(update)它。然而，你可以把一个标记了`[DisableAutoCreation]`属性的system添加到一个`ComponentSystemGroup`中，这个system会自动地被更新，就像没有标记此属性的systems一样。

3.4.3 Default System Groups

Default World包含有一群具有层次结构的`ComponentSystemGroup`的实例。仅有三个根层次(root-level)的system groups被添加到了Unity player loop (下面的列表也显示了每个group中预定义的systems):

- InitializationSystemGroup (在player loop的`Initialization`阶段的末尾被更新)
    - BeginInitializationEntityCommandBufferSystem
    - CopyInitialTransformFromGameObjectSystem
    - SubSceneLiveLinkSystem

        - SubSceneStreamingSystem
        - EndInitializationEntityCommandBufferSystem
    - SimulationSystemGroup (在player loop的`Update`阶段的末尾被更新)
        - BeginSimulationEntityCommandBufferSystem
        - TransformSystemGroup
            - EndFrameParentSystem
            - CopyTransformFromGameObjectSystem
            - EndFrameTRSToLocalToWorldSystem
            - EndFrameTRSToLocalToParentSystem
            - EndFrameLocalToParentSystem
            - CopyTransformToGameObjectSystem
        - LateSimulationSystemGroup
        - EndSimulationEntityCommandBufferSystem
    - PresentationSystemGroup (在player loop的`PreLateUpdate`阶段的末尾被更新)
        - BeginPresentationEntityCommandBufferSystem
        - CreateMissingRenderBoundsFromMeshRenderer
        - RenderingSystemBootstrap
        - RenderBoundsUpdateSystem
        - RenderMeshSystem
        - LODGroupSystemV1
        - LodRequirementsUpdateSystem
        - EndPresentationEntityCommandBufferSystem

(注意，此列表的具体内容可能会更改, ECS还处于Preview阶段)

3.4.4 Multiple Worlds

除了default World外，你还可以创建多个Worlds。同一个component system class可以在多个World中被实例化，并且每一个实例可以在update order中的不同点以不同的速率进行更新。

目前还没有办法手动的update一个给定World中的每一个system；除非，你可以控制哪些systems在哪个World中被创建，以及它们(systems)被添加到哪些现存的system groups。例如，一个自定义的WorldB可以实例化SystemX和SystemY，添加SystemX到default World的`SimulationSystemGroup`，添加SystemY到default World的`PresentationSystemGroup`。这些systems也可以在组内相对于其它systems(指自动创建的)进行排序，也会随着相应的group更新。

为了支持这种用例，可以使用`ICustomBootstrap`接口:

```
public interface ICustomBootstrap
{
    // Returns the systems which should be handled by the default bootstrap process.
    // If null is returned the default world will not be created at all.
    // Empty list creates default world and entrypoints
    List<Type> Initialize(List<Type> systems);
}
```

当你的类实现这个interface时，完整的component system types列表将会在default world initialization之前被传入到你的类的`Initialize()`方法。一个自定义的bootstrapper可以遍历这个列表，并且在任何你想的World中创建systems。你可以在`Initialzie()`方法中返回systems列表，它们会在default world initialization中被创建。

举例，下面是一个典型的自定义的`MyCustomBootstrap.Initialize()`的实现:

1. 创建任何额外的Worlds (这里应该指default world之外的worlds)和它们的顶层(top-level) ComponentSystemGroups
2. 对于在system Type列表中的每一个Type:
    a. 向上遍历ComponentSystemGroup的层次结构(hierarychy)，找到这个system type所属的顶层group。
    b. 如果这个顶层group是step1中创建group，在那个对应的World中创建这个system，并把它添加到这个层次结构中(使用`group.AddSystemToUpdateList()`)。
    c. 如果不是，把这个system Type添加到要返回的List中，由DefaultWorldInitialization处理。
3. 在新建的top-level groups上调用`group.SortSystemUpdateList()`
    a. 可选择性地添加它们到某个default world group中。
4. 返回未处理的systems列表到DefaultWorldInitialization。

注意: ECS框架通过反射找到你的`ICustomBootstrap`的实现。

3.4.5 Tips and Best Practices

- **为你写的每一个system，用[UpdateInGroup]指定system group**。如果不指定，默认会使用SimulationSystemGroup。
- **使用手动标记的 ComponentSystemGroups来更新systems (在游戏循环的其它地方, elsewhere in the Unity player loop)**。给一个component system 或system group添加[DistableAutoCreation]属性，可以阻止它被创建或添加到default system groups。你依然可以在主线程中，手动地使用`World.GetOrCreateSystem()`来创建这个system，手动地调用`MySystem.Update()`来更新它。这是一个在Unity player loop的其它 地方插入systems的简单方法 (例如，你有一个system，想让它帧中提前和稍后运行)。
- **可能的话，要使用已存在的`EntityCommandBufferSystem`s，而不是添加新的。** 一个`EntityCommandBufferSystem`表示一个同步点(sync point): 在处理任务未处理的`EntityCommandBuffer`之前，主线程为在这里等待所有工作线程(worker threads)完成。
- **避免把自定义逻辑放在`ComponentSystemGroup.OnUpdate()`**。因为`ComponentSystemGroup`本身在功能上就是一个component system，向它的`OnUpdate()`方法中添加自定义处理来完成一些工作、产生一些jobs等是很吸引人的。但是我们建议不这样干，因为从外部看，还不太清楚它的自定义逻辑是在组成员更新前还是更新后执行。最好仅仅使用system groups的分组功能，而在独立的component system中实现逻辑。

4 Accessing entity data (访问实体数据)

在实现有一个ECS系统时，你需要经常遍历你的数据。ECS系统处理一堆entities时，通常是从一个或多个component中读取数据，然后完成计算，最后把结果写入到另一个component中。

总的来说，遍历entities和components最有效的方法是在一个可并行(parallelizable)的job中，按顺序处理components。这样可以充分利用所有可用cpu cores的计算能力，利用数据局部性(data locality)来避免CPU 缓存丢失(cache misses)。

ECS API 提供了许多方法来完成遍历，每一个方法都有它自己的性能影响和限制。你可以用以下方法来遍历ECS data:

- IJobForEach: 按实体处理组件数据的最简单有效方法。
- IJobForEachWithEntity: 比IJobForeach稍微复杂点的方法，给了你访问entity handle的权限，以及访问你正在处理的enity的数组索引的权限。
- IJobChunk: 遍历Chunk，即包含有符合条件的entities的内存块。你的Job的`Execute()`函数可以使用for循环在每一个Chunk上迭代遍历。
- ComponentSystem: `ComponentSystem`提供了`Entities.ForEach`委托函数来帮助迭代遍历entities。然而，`ForEach`在主线程上运行。
- Manual iteration: 如果上面提到的几种方法任不够用，你可以手动迭代遍历你的entities或chunks。例如，你可以获取一个包含有你想要处理的entities的NativeArray(或chunks)，然后使用一个Job，如：IJobParallelFor，来遍历这个NativeArray(或chunks)。

EntityQuery类提供了一个构建数据视图的方法。它构建的视图中只包含有你的算法想要的确定数据。这类似于数据库中的`Create View`操作。上面列举的迭代方法大部分都明显地或在内部使用了EntityQuery。

5 Jobs in ECS

ECS使用Job system来实现行为(behavior)，即ECS中的*System*。一个ECS系统具体来说是一个Job，用于变换存储在entity components中的数据。

例如，下面的system更新位置：

```
1  using Unity.Burst;
2  using Unity.Collections;
3  using Unity.Entities;
4  using Unity.Jobs;
5  using Unity.Transforms;
6  using UnityEngine;
7
8  public class MovementSpeedSystem : JobComponentSystem
9  {
10     [BurstCompile]
11     struct MovementSpeedJob : IJobForEach<Position,
   MovementSpeed>
12     {
13         public float dT;
14
```

```
15          public void Execute(ref Position Position,
    [ReadOnly] ref MovementSpeed movementSpeed)
16          {
17              float3 moveSpeed = movementSpeed.Value *
    dT;
18              Position.Value = Position.Value +
    moveSpeed;
19          }
20      }
21
22      // OnUpdate runs on the main thread.
23      protected override JobHandle OnUpdate(JobHandle
    inputDependencies)
24      {
25          var job = new MovementSpeedJob()
26          {
27              dT = Time.deltaTime
28          };
29
30          return job.Schedule(this, inputDependencies);
31      }
32  }
```

Job extensions

Unity C# Job System让你的代码在多线程上运行。它提供了任务调度，并行处理，和多线程安全。Job System是一个核心的Unity模块，它为创建和运行jobs提供了通用的接口和类(不管你用不用ECS)。这些接口包括：

- IJob: 创建一个运行在任何线程或CPU core上的Job (由Job System的调度程序决定)。
- IJobParallelFor: 创建一个可并行地运行在多个线程上的Job，用于处理NativeContainer的元素。
- IJobExtensions: 为运行`IJobs`提供了扩展方法。
- IJobParallelForExtensions: 为运行`IJobParallelFor`类型的jobs提供了扩展方法。
- JobHandle: 一个访问被调试job的句柄(handle)。`JobHandle`实例允许你说明Jobs之间的依赖关系。

C# Job System是独立于ECS的，不用ECS时也能使用它。

Jobs package扩展了Job System来支持ECS:

- IJobParallelForDeferExtensions
- IJobParallelForFilter
- JobParallelIndexListExtensions
- JobStructProduce

References:

- ECS Manual
- EntityComponentSystemSamples