Entitas-CSharp源码解析

seanhu 2019.10

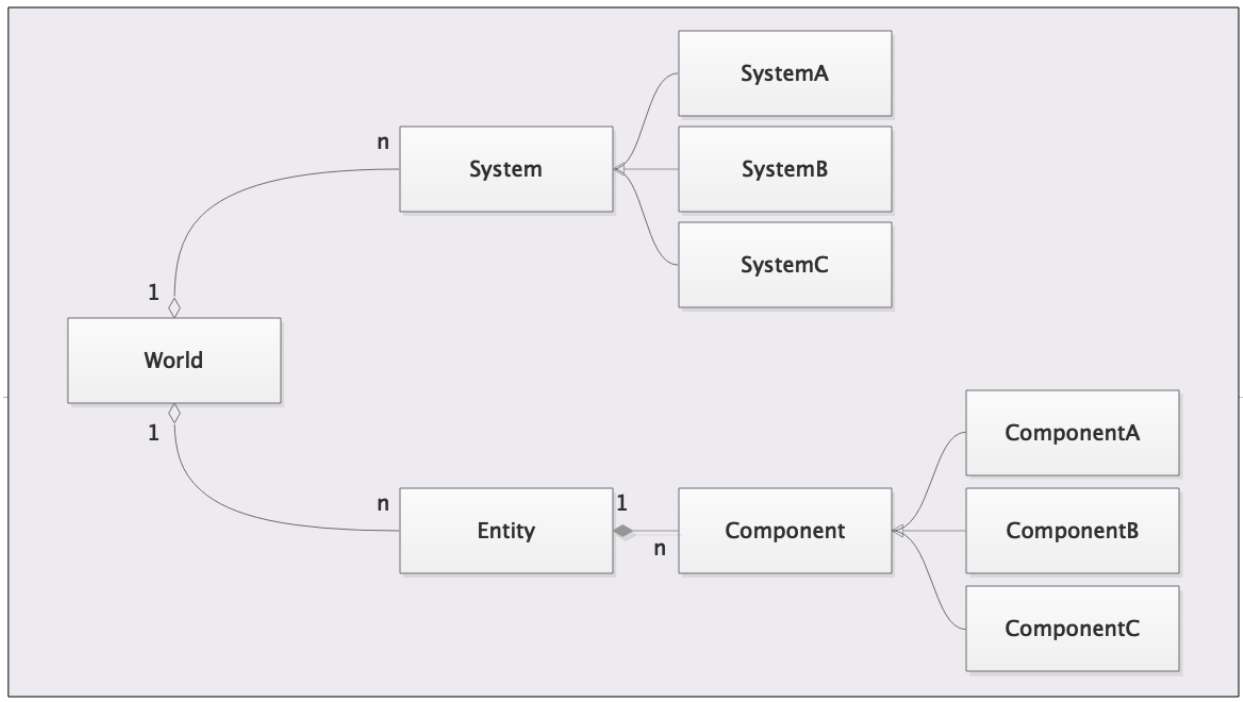
简介

Entitas是一个快速、轻量的ECS（Entity-Component-System）框架。最早是以C#语言实现并专门设计用于Unity引擎，后来衍生出了很多其他语言版本，并且在非Unity项目中也有应用。

本文将基于C#版的Entitas进行源码解析，主要基于国人改造过的No-CodeGenerator版本。项目地址如下：
CodeGenerator版本： https://github.com/sschmid/Entitas-CSharp
No-CodeGenerator版本：https://github.com/rocwood/Entitas-Lite

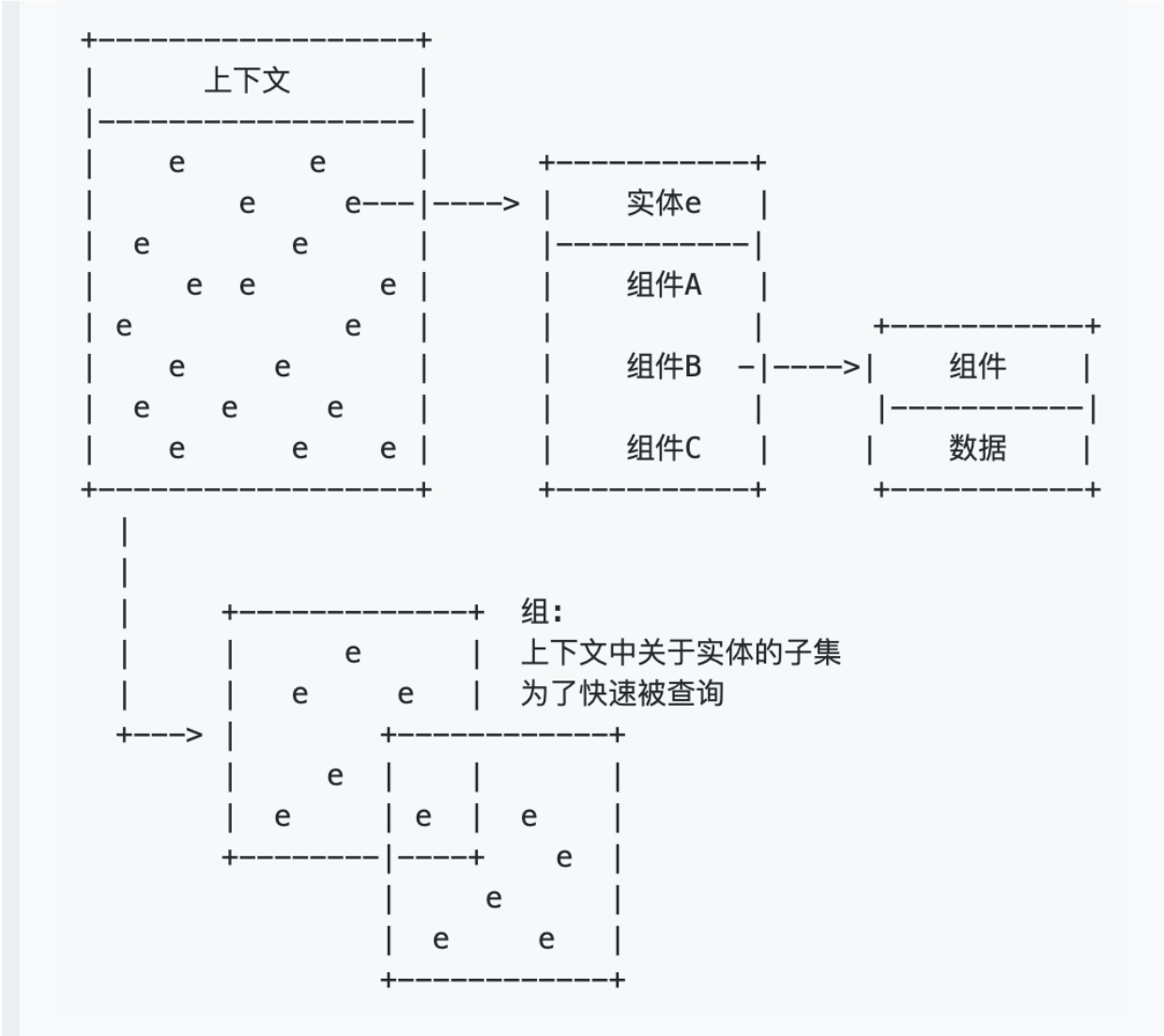
ECS架构

ECS架构如下图所示，是一种面向数据思想的框架，其最显著特点是Component只用来存储数据且没有任何行为，System有行为但没有数据。



ECS框架可以更好地分解复杂问题、整理复杂的关系；添加新的行为代码变得非常明确；减少内部系统之间的耦合，降低开发复杂度；有利于做数据快照，数据回滚；对CPU缓存极其友好。

Entitas中几个重要概念一览：



源码解析

Entity

entity对应应用中的具体对象（比如游戏世界中的Actor，PlayerController，Inventory等），它是一个组合了不同Component的容器，通过增加、替换或删除Component来改变实体中的数据，同时Entitias具有相应的事件，可以知道是否添加、替换或删除了组件。

Automatic Entity Reference Counting（AERC）

Entity的引用计数是一个内部机制，用于防止将还在使用的Entity实例（当前还被其他对象引用）放入重用对象池中（引用计数为0时放入）。

当需要保存一个Entity实例的引用时，必须手动调用entity.Retain(this)来增加Entity的引用计数，不再需要保留的时候必须再手动调用entity.Release(this)来减少Entity的引用计数。所有Entitas内部类都已经处理了Entity的这个引用计数机制。

如果不调用Retain，则持有的Entity实例可能在某个时候被Destroy掉并且被重用做一个新的Entity实例，从而造成逻辑异常。如果忘记调用Release，那这个Entity实例永远不会被Destroy，造成逻辑上

的内存泄漏。

Entitas内部实现了两个AERC类，一个是SafeAERC，另一个是UnSafeAERC。二者的区别是SafeAERC内部使用HashSet来记录引用了该Entity的对象，Retain操作会检查是否重复，重复则抛出异常。Release操作会检查是否已Retain过，如果没有则抛出异常；UnsafeAERC内部仅使用一个整形变量记录引用数，无法知道是哪些对象引用了该Entity，Retain和Release操作也不做任何检查，性能要比SafeAERC高。默认是使用SafeAERC。

实作技巧：由于允许外部定制AERC的实现，所以IEntity继承了IAERC接口，同时Entity类中定义了一个IAERC属性，在Entity的Initialize函数中可以传入定制的AERC实例。（UML类图中类之间的关系怎么画？）

Entity创建和销毁

一个Entity必须由context负责其生命期，不能直接使用new实例化一个Entity实例（如何从设计上避免误用？），需要通过context.CreateEntity()来创建（更多细节参见Context章节部）。销毁一个Entity实例的正确姿势是调用entity.Destroy()，通过事件机制最终会调用到context.onDestroyEntity()来启动Entity的销毁流程（更多细节参见Context章节）。

需要注意的是Destroy一个Entity并不会真的销毁，而是放入到了一个对象池中，这是性能优化和避免GC的一个设计。

事件

Entity中可被订阅的事件包括：

- OnComponentAdded: 当增加一个Component时触发调用。
- OnComponentRemoved：当移除一个Component时触发调用。
- OnComponentReplaced：当一个Component被替换（同类型替换）时触发调用。
- OnEntityReleased：执行Release操作后，引用计数变为0时触发调用。
- OnDestroyEntity：执行entity.Destroy()时触发调用。

当Entity实例被context销毁时，该Entity所有的event handlers都会被移除。

属性、成员变量

- **totalComponents**：entity可以包含Component的最大数目，由context在初始化entity时传入。该值决定了IComponent[] _components数组的大小。
- **creationIndex**：entity的唯一索引，也是EntityEqualityComparer.GetHashCode的返回值，context创建Entity时设置。context中也会以该索引值作为key维护一个字典（Dictionary<int, Entity> _entitiesLookup），方便根据索引值快速查询Entity实例。注意只能保证同一个context下Entity的该值是唯一的，为了保证整个应用该值的全局唯一，我们可以给每个context分配不同的索引值区间段。
- **isEnabled**：entity的状态，已经destroyed的entity该值为false。
- **Stack[] componentPools**：用于已回收component的重用，由context在创建entity时设置。也是基于性能和避免GC的一个设计。
- **contextInfo**：包含context的一些基本信息（名称、component名称等），context创建entity时设置，主要用于提供更明确的错误日志。

- **_components，_componentsCache，_componentIndicesCache**：_components数组长度固定为_totalComponents，数组下标为component类型的Index值；_componentsCache是entity实际包含的component列表。_componentIndicesCache是entity实际包含的component类型index列表。后面这两个cache变量都是通过前面的_components变量重建出来的。

Component相关操作

- **AddComponent(int index, IComponent component)**：增加一个component。entity如果处于非Enable状态（即destroyed）将抛出异常。同一个index（每一种component类型对应一个唯一的index）下只能有一个component，否则也会抛出异常。增加component成功后会触发event OnComponentAdded事件调用。
- **RemoveComponent(int index)**: 移除指定index的component。内部实现上是调用了replaceComponent。entity处于非enable状态或者指定index的component不存在则会抛出异常。移除component成功后触发OnComponentRemoved调用，最后会重置该component并放入回收重用对象池中。
- **ReplaceComponent(int index, IComponent component)**: 将指定index的component替换掉，如果指定index原来没有component，则等同于AddComponent。执行成功后将触发OnComponentReplaced事件调用。
- **查询**：GetComponent，GetComponents，GetComponentIndices，HasComponent，HasComponents，HasAnyComponent。这些查询类函数均不会触发异常。
- **创建**：优先使用回收复用对象池中的对象，如果没有可复用的才会新创建。注意，这里仅仅是创建component，并不会加到entity的component集合中，这是因为新创建的component往往还需要做额外的初始化工作，初始化完成后再调用AddComponent增加到entity中。

EntityEqualityComparer

这里涉及到C#中使用IEqualityComparer和重写类本身的Equals / GethashCode的区别。

重写Equals和GetHashCode是改变类本身的行为，而且如果不同时重写==运算符，则使用该运算符比较对象将不会与Equals具有相同的行为。

使用IEqualityComparer允许更多的灵活性，可以实现通用的解决方案。如果其他类也有相同的比较逻辑，可以不需要在所有类中都重写Equals和GetHashCode。

No-CodeGenerator版本对Entity类的扩展

CodeGenerator版本的Entitas，Entity，Context，Contexts，Matcher，Feature是会由CodeGenerator根据所属context自动生成多个类的，比如GameEntity，GameContext，GameMatcher，InputEntity…。No-CodeGenerator版本则只会有一个实现类。

CodeGenerator版本Entity类与Component相关的API接口都需要传入component类型的index，这些细节可以由CodeGenerator自动生成的代码隐藏。

No-CodeGenerator版本增加一组新的Component操作接口，可以不用关心component-index（根据component类型获取其index的细节参见component章节）。具体的实现代码在EntitiGeneric.cs文件中。新增的这组接口允许使用者决定在增加component的时候是否返回已经存在的实例而不抛出异常；在移除component的时候是否忽略不存在的错误；同时提供了一个标记component已被修改而触发GroupEvent和ReactiveSystem。

Component

Component在Entitas中是最简单的实现部分，只定义了两个简单的接口类（因为ECS中的Component本来就只有数据没有行为）。

```
public interface IComponent {
    }

public interface IUniqueComponent : IComponent {
    }
```

实际应用项目中可以根据需要扩展出更多的Component类型，比如：

- **The simplest Component - Flag Component**: 不含任何属性。
- **Data Component**：包含一个或多个属性，强调存储的是纯数据。
- **Reference Component**：类似Data Component，区别是其属性不表示数据，而是引用复杂的对象。
- **Action Component**：属性是function/action。
- **Unique Component**：标记为唯一，在整个应用中只能有一个实例。

Component-Index

每一个Component类都从属于context，并且在一个context内有一个唯一索引。
同一个Component类是允许出现在多个context中的，通过添加多个类注解（annotate）来实现。
在不同的context里这个唯一索引值互相之间没有关系，是可以不相等的。

具体的代码实现参见Context章节。

一个应用中需要多少Component类?

取决于实际项目，一般来说对于一个中等复杂度的游戏项目来说，将这个数量控制在150个左右是比较合理的。

由于一个context里Component类的数量决定了entity的内存占用大小（Entity._components数组变量），所以需要将Component类合理地划分到不同的Context。

Matcher

Matcher用于描述哪些entities是我们关心的，为Group提供查询entities的能力。

- AllOf：包含了所有指定components的entities。
- AnyOf：包含了指定components中任何一个即可。
- NoneOf：不能包含指定components中的任何一个。
  这些表达式可以组合使用，特别需要注意的是NoneOf不要单独使用，因为这可能返回一个巨大的entities列表。

举例
No-CodeGenerator版:
var matcher = Matcher.AllOf<PositionComponent, VelocityComponent>();

CodeGenerator版本:
context.GetGroup(GameMatcher.AllOf(GameMatcher.Position,
GameMatcher.Velocity).NoneOf(GameMatcher.NotMovable));

注意：No-CodeGenerator由于没有自动生成代码的辅助（不能便利地获取Component-index），所以写链式组合式的Matcher相对复杂一些。

类图

IMatcher <— ICompoundMatcher <— INoneOfMatcher <— IAnyOfMatcher <— IAllOfMatcher <— Matcher

初次看这部分源码的时候，这些接口类的继承关系令人费解，细细研究后才发现其实这是为了通过链式串联AllOf，AnyOf，NoneOf来实现更复杂的查询。

实现

内部是存储了三个component-index的数组，然后通过判断指定index位置的component是否存在来实现。

```
    int[] allOfIndices { get; }
    int[] anyOfIndices { get; }
    int[] noneOfIndices { get; }

public bool Matches(Entity entity) {
    return (_allOfIndices == null || entity.HasComponents(_allOfIndices))
        && (_anyOfIndices == null || entity.HasAnyComponent(_anyOfIndice
s))
        && (_noneOfIndices == null || !entity.HasAnyComponent(_noneOfIndi
ces));
}
```

CodeGenerator版本的Matcher是设计成了泛型类，这是因为这个版本在不同的context下有不同的Entity类（我觉得没必要设计成泛型类？）。
No-CodeGenerator版本是设计成普通类。

No-CodeGenerator版本为了使创建Matcher变得更简单，实现了一组静态模板类（MatcherGeneric.cs）。

疑问：下面代码片段中为什么有indices.Length == 1的限制？应用场景？

```
static int[] mergeIndices(IMatcher<TEntity>[] matchers) {
    var indices = new int[matchers.Length];
    for (int i = 0; i < matchers.Length; i++) {
```

```
            var matcher = matchers[i];
            if (matcher.indices.Length != 1) {
                throw new MatcherException(matcher.indices.Length);
            }
                indices[i] = matcher.indices[0];
        }

        return indices;
    }
```

Group

在ECS框架中，System会关心那些拥有指定components的entities，由于context管理了所有的entities，通过遍历是可以获取到system关心的entities的，但是这样效率非常低。为了解决这个问题，引入了Group的概念。

Group是一个包含了满足一定条件的entities容器，Matcher决定哪些entities可以加到group中，group中的entities列表会实时更新。

context内部管理了一个可重用的group列表，相同的matcher将返回相同的group，因此频繁访问含有相同matcher的group不会有额外的开销。

事件

OnEntityAdded：当entity添加到group中时触发调用。
OnEntityRemoved：当entity从group中移除时触发调用。
OnEntityUpdated：当group中的entity被更新时（即entity的一个component被替换时）触发调用。

属性，成员变量，成员函数

HashSet _entities: group中保存的entities。
Entity[] _entitiesCache: 以数组形式返回group中的entities，根据上面的_entities变量动态生成。
Entity _singleEntityCache:
HandleEntitySilently: 静默处理entity，不会触发事件回调，不会触发异常。如果满足matcher条件，则将entity添加到group列表中（内部调用addEntitySilently，如果已经存在则什么也不做），如果不满足，则从group列表中移除（内部调用removeEntitySilently，如果不存在则什么也不做）。
HandleEntity: 与HandleEntitySilently对应，内部调用addEntity和removeEntity。在成功执行后会触发事件调用（OnEntityAdded，OnEntityRemoved）。

另外一个重载函数：**GroupChanged HandleEntity(Entity entity)**，这个函数存在的意义？应用场景？我觉得可以不需要。

UpdateEntity：在该函数中会依次触发OnEntityRemoved，OnEntityAdded，OnEntityUpdated事件调用。
内部实现并不会真的移除后再增加component，而是先模拟触发remove component事件（此时

component是旧值），然后再设置component新值，最后再触发add component事件。（No-CodeGenerator版在那实现的该逻辑？）

GetSingleEntity：如果group为空，返回null，如果entity个数大于1则抛出异常，只有当entity个数为1时才能正确返回这个唯一的entity实例。
疑问：下面代码片段为什么要使用using?

```
public Entity GetSingleEntity() {
    if (_singleEntityCache == null) {
        var c = _entities.Count;
        if (c == 1) {
            using (var enumerator = _entities.GetEnumerator()) {
                enumerator.MoveNext();
                _singleEntityCache = enumerator.Current;
            }
        } else if (c == 0) {
            return null;
        } else {
            throw new GroupSingleEntityException(this);
        }
    }
    return _singleEntityCache;
}
```

Collector

根据指定的GroupEvent（Add，Remove，AddOrRemove）监测groups（这些groups必须在同一个context下）中发生变化的entities，处理完后要调用ClearCollectedEntities清理掉收集到的entities。

需要注意的是，当我们监测group.Remove，一个entity被收集到了collector，稍后如果这个entity又满足group的matcher条件重新加回到group中，此时这个entity依然会保留在collector中。

collector可以被激活和去激活。

内部实现其实就是根据传入的group.event去订阅group的OnEntityAdded、OnEntityRemoved事件。

属性，变量

HashSet collectedEntities：存储收集到的entities，处理完后需要手工清理该集合的数据。

GroupChanged _addEntityCache;：为什么需要增加这个cache变量？

Monitor

monitor是构建在collector之上的对象，可以设置filter和processor委托。

Execute处理流程：对collector收集的entities进行过滤，符合条件的entitie暂存到_buffer集合中；清空collector的entities列表；对_buffer集合中每一个entity执行_processor委托；清空_buffer临时集合。

CodeGenerator版本没有Monitor。
MonitorList类我觉得可以去掉。

Systems

ECS的主要目标是分离状态和行为。System就是我们定义行为的地方。

在Entitas中定义了几种类型的System。
接口继承关系
ISystem <— ICleanupSystem
ISystem <— IExecuteSystem <— IReactiveSystem
ISystem <— IInitializeSystem
ISystem <— ITearDownSystem

- **InitializeSystem**
  仅执行一次的系统，用于实现整个应用的初始化逻辑（Initialize函数）。
- **CleanupSystem**
  定期执行的系统，是在所有的ExecuteSystem执行完毕后执行（Cleanup函数）。
- **TearDownSystem**
  仅执行一次的系统，用于实现整个应用的结束逻辑（TearDown函数）。
- **ExecuteSystem**
  定期执行的系统，每次tick时调用（Execute函数）。内部是存储了一个matcher对象，根据matcher从context中获取满足条件的entities，然后遍历每个entity执行操作（纯虚函数，子类负责实现具体逻辑）。
- **ReactiveSystem**
  变化时触发执行的系统。内部存储了一个monitor对象列表，Activate()函数激活所有monitor，开始监控entity的变化。Deactivate()函数去激活所有monitor，停止监控entity的变化（此时monitor关联的collector收集的entities数量一直为0）。Execute()函数会在tick时定期执行，如果monitor关联的collecor收集到了变化的entities，则遍历这些entities执行monitor指定的操作（_processor委托）。
- **Composing systems**
  将前面介绍的system类型（InitializeSystem, CleanupSystem, TearDownSystem, ExecuteSystem(含ReactiveSystem)）组合在一起使用。
  内部是记录了四个system集合，按照加入集合的顺序执行。注意，集合中的system可以内嵌其他composing systems的，从ActivateReactiveSystem()可以看出来。

```
protected readonly List<IInitializeSystem> _initializeSystems;
protected readonly List<IExecuteSystem> _executeSystems;
protected readonly List<ICleanupSystem> _cleanupSystems;
protected readonly List<ITearDownSystem> _tearDownSystems;

public void ActivateReactiveSystems() {
    for (int i = 0; i < _executeSystems.Count; i++) {
```

```
            var system = _executeSystems[i];
            var reactiveSystem = system as IReactiveSystem;
            if (reactiveSystem != null) {
                reactiveSystem.Activate();
            }

            var nestedSystems = system as Systems;
            if (nestedSystems != null) {
                nestedSystems.ActivateReactiveSystems();
            }
        }
    }
```

Feature

Feature继承自Systems类，是为了简化对system的管理而设计的，可以根据指定的名称从当前程序域获取所有的system类（实现了ISystem接口），并按照属性标签(FeatureAttribute)标记的优先级进行排序，再按序创建出system实例加入到Systems的各个集合中（_initializeSystems，_executeSystems，_cleanupSystems，_tearDownSystems）。

未用FeatureAttribute标记的system将放入UnnamedFeature。

注意：Feature不支持嵌套，即不会自动收集继承自Systems的类。

举例：
var _feature = new Feature(“Game”);
创建了一个Composing systems，包含了所有使用属性标签[FeatureAttribute(“Game”, 0)]标记过的system。

Context

context负责管理entities和groups的生命期。一个应用中可以有多个context。

事件

OnEntityCreated
OnEntityWillBeDestroyed
OnEntityDestroyed

OnGroupCreated

属性，变量

- **Stack[] componentPools** 数组下标是component-index，用于component对象的重用，同时避免了自动GC。
- **HashSet _entities**： entity实例集合。

- **Stack _reusableEntities**： 可被重用的entities集合。
- **HashSet _retainedEntities**：返回当前还被其他对象（比如Group，Collector，ReactiveSystem等）retained的entities集合。
- **Dictionary<string, IEntityIndex> _entityIndices**：
- **Dictionary<IMatcher, IGroup> _groups**：以matcher作为key缓存所有创建过的group。
- * *List[] _groupsForIndex**：数组下标为component-index，用于缓存与指定component-index相关的group列表。
- **IGroup[] _groupForSingle**：数组下标为component-index，用于缓存unique-component对应的group。
- **Dictionary<int, Entity> _entitiesLookup**：以entity的creation-index作为key缓存的entity实例集合。
- **int _creationIndex**：每创建一个entity该值加1.
-

函数

- **CreateEntity**
  创建一个新的entity。
  1、优先复用_reusableEntities集合中的实例，并调用entity.Reactivate()来重新激活entity和重新赋值creationIndex。
  2、放入_entities集合
  3、订阅entity事件
  3、触发事件OnEntityCreated
- **DestroyEntity**
  销毁指定entity，移除entity的所有component实例并放入重用池中。注意不要直接调用该函数，而是应该使用entity.Destroy()来销毁entity。
  1、从_entities集合中移除
  2、触发调用OnEntityWillBeDestroyed
  3、entity.InternalDestroy：移除所有component，置_isEnabled为false，重置除OnEntityReleased以外的其他事件订阅。
  4、触发调用OnEntityDestroyed
  5、如果还有其他对象引用了该entity，则只是简单减少引用计数，并放入_retainedEntities集合
  6、如果没有其他对象引用该entity，则放入_reusableEntities集合，
- **IGroup GetGroup(IMatcher matcher)**
  根据指定的matcher返回group实例。如果是第一次获取指定matcher的group，则按下面步骤创建：
  1、创建group实例
  2、遍历context下的entities集合，将所有满足matcher条件的加入到group中
  3、group实例放入到_groups集合
  4、group实例放入_groupsForIndex集合
  5、触发调用OnGroupCreated
- **UniqueComponent相关的一组函数**
  GetSingleEntity
  GetSingleEntty
  GetUnique
  GetUniqueComponent
  AddUnique

ModifyUnique
ModifyUniqueComponent

- **updateGroupsComponentAddedOrRemoved**
  所有entity实例的OnComponentAdded和OnComponentRemoved事件回调的时候都会调用到该函数。
  1、根据component-index从_groupsForIndex集合中直接找到与此component相关的groups，然后遍历这些group以决定entity是否应该从group中移除或添加。

ContextAttribute

context属性标签类，用于标记component属于哪个context。

ContextInfo

每一个context对应一个contextInfo实例，存储了context的名称、context下所有component类型的名称和类型信息。

另外，提供了一个根据component类型信息查询其Index的接口，这个Index是Component的唯一索引值，在Entitas中有大量使用。

ComponentIndex

以模板类实现，可以快速根据component类型查询到其Index。第一次查询的时候是调用ContextInfo的查询接口，之后就直接使用缓存值。

注意：同一个Component是可以出现在多个context下的，在不同context下的component-index之间没有任何关系。

Contexts

一个单例，管理应用中所有的context。

初始化context

1、收集程序域中所有Component类（实现了IComponent接口），根据类的属性标签值（ContextAttribute）生成一个字典集合（Dictionary<string, List>），即每个context下定义的component类型列表。没有定义ContextAttribute的Component将放入一个默认Context中。
2、创建context实例，放入Dictionary<string, Context> _contextLookup; 这样就可以根据context名称快速查找到context实例。

查询context

1、根据contextName查询
2、根据ContextAttribute查询，以模板函数实现，最终也是取属性标签的Name进行查询。

EntityIndex

EntityIndex与Group紧密相关。我们创建一个Group来获取所有含PositionComponent的entities，如果希望进一步获取在指定位置的entities，则可以建立一个EntityIndex来实现。
EntityIndex在内部实现上是一个Group的Observer，通过订阅Group事件（OnEntityAdded，OnEntityRemoved）来生成索引。

在Entitas中有两种类型的索引：

- PrimaryEntityIndex: 一个Key对应一个Entity，内部数据结构：Dictionary<TKey, Entity> _index。
- EntityIndex：一个Key对应一组Entity，内部数据结构：Dictionary<TKey, HashSet> _index。

在实际项目中可以根据需要再扩展出更复杂的Index。

如何使用

定义Component类

```
[Game]
public class PositionComponent : IComponent
{
    public int x;
    public int y;

    public void SetValue(int nx, int ny)
    {
        x = nx;
        y = ny;
    }
}
```

1、component只有数据，没有行为。但是可以定义helper accessor；
2、使用属性标签标注component属于哪个context，如果未标记则属于Default context；

定义System类

1、system只有行为，没有数据；
2、使用属性标签标注system属于哪个Feature，如果未标记则属于UnnamedFeature

```
public class MoveSystem : IExecuteSystem
{
    public void Execute()
```

```
    {
        var entities = Context<Default>.AllOf<PositionComponent, Velocity
Component>().GetEntities();
        foreach (var e in entities)
        {
            var vel = e.Get<VelocityComponent>();
            var pos = e.Modify<PositionComponent>();
            pos.x += vel.x;
            pos.y += vel.y;
        }
    }
}
```

创建entity

创建Group