PoolManager

ihaiu@qq.com

Class

PoolManager	总入口类
PoolGroupDict	PoolManager的延伸，保存PoolGroup的字典。
PoolGroup	对象池组，保存Pool、Spawn实例对象、Despawn实例对象。
ObjectPool<T>	对象池。
ScriptableObjectPool<T>	继承自ObjectPool<T>。如果是继承了ScriptableObject的类推荐使用这个。
PrefabPool	继承自ObjectPool<T>。预设的对象池。
IPoolItem	实例对象接口，自定义的类推荐实现它。

PoolManager

```
/** PoolGroup字典 */
public static PoolGroupDict groups = new PoolGroupDict();
```

PoolManager.groups

```
 /** 创建PoolGroup */
 public PoolGroup Create(string groupName)

 /** 销毁PoolGroup */
 public bool Destroy(string groupName)

 /** 销毁所有PoolGroup */
 public void DestroyAll()


 /** 是否已经存在groupName的PoolGroup */
public bool ContainsKey(string groupName)

 /** 尝试获取groupName的PoolGroup */
public bool TryGetValue(string groupName, out PoolGroup poolGroup)

 /** 获取key的PoolGroup */
public PoolGroup this[string key]



/** 默认PoolGroup，可以自己扩展 */
public PoolGroup common
{
            get
            {
                if (!ContainsKey("CommonPoolGroup"))
                {
                    Create("CommonPoolGroup");
                }

                return this["CommonPoolGroup"];
            }
}
```

为了方便调用，扩展一个常用PoolGroup

```
namespace Ihaius
{
    public partial class PoolGroupDict
    {
        /** 战斗对象池组 */
        public PoolGroup war
        {
            get
            {
                if (!ContainsKey("WarPoolGroup"))
                {
                    Create("WarPoolGroup");
                }

                return this["WarPoolGroup"];
            }
        }
    }
}
```

PoolGroup属性

```
/** 组名称 */
public string groupName;

/** 是否打印日志信息 */
public bool logMessages = true;

/** 检测粒子设为闲置状态最长时间 */
public float maxParticleDespawnTime = 300;
```

PoolGroup方法

```
 /** 添加一个对象池 */
public void CreatePool<T>(ObjectPool<T> objectPool)

/** 获取对应类型的Pool */
public ObjectPool<T> GetPool<T>()
public AbstractObjectPool GetPool(Type type)


 /** 获取一个对应类型的实例对象 */
 public T Spawn<T>(params object[] args)


 /** 将实例对象设置为闲置状态 */
public void Despawn<T>(T inst)

 /** 延迟seconds秒，将实例对象设置为闲置状态 */
public void Despawn<T>(T instance, float seconds)



 /** 添加一个实例到对应的Pool。despawn:true为闲置状态;false为真正使用状态 */
 public void Add<T>(T instance, bool despawn)
```

PoolGroup用Prefab的方法

```
 /** 添加一个对象池 */
public void CreatePool(PrefabPool prefabPool)

/** 获取对应预设的Pool */
public PrefabPool GetPool(Transform prefab)
public PrefabPool GetPool(GameObject prefab)

/** 获取预设*/
public Transform GetPrefab(Transform instance)
public GameObject GetPrefab(GameObject instance)
```

```
// 添加实例对象instance到名称为prefabName对象池
// despawn,true为闲置状态;false为真正使用状态,并把该对象添加到SpawnPool的_spawned列表
// parent, true时instance.parent=group
public void Add(Transform instance, string prefabName, bool despawn, bool parent)


// 检测声音停止播放协程
// 如果声音停止播放了就调用Despawn。设置该对象为闲置状态
private IEnumerator ListForAudioStop(AudioSource src)
// 检测粒子生命结束协程
// 如果粒子gameObject没激活就取消检测。emitter.IsAlive(true)==false时就调用Despawn。设置该对象为闲置状态
// 如果检测时间超过maxParticleDespawnTime就抛出一个警告
private IEnumerator ListenForEmitDespawn(ParticleEmitter emitter)
private IEnumerator ListenForEmitDespawn(ParticleSystem emitter)
```

```
// 从对象池获取一个实例对象或者创建一个实例对象
// 检测是否存在prefab的对象池
// 如果存在，就从prefabPool里拿一个实例对象。如果这个实例对象是null就返回null；否则就设置pos, rot。parent为空就设置为group。最后在该gameObejct广播OnSpawned消息
// 如果不存在,就创建一个PrefabPool。然后和上面一样的步骤
public Transform Spawn(Transform prefab, Vector3 pos, Quaternion rot, Transform parent)
public Transform Spawn(Transform prefab, Vector3 pos, Quaternion rot)
public Transform Spawn(Transform prefab)
public Transform Spawn(Transform prefab, Transform parent)
public Transform Spawn(string prefabName)
public Transform Spawn(string prefabName, Transform parent)
public Transform Spawn(string prefabName, Vector3 pos, Quaternion rot)
public Transform Spawn(string prefabName, Vector3 pos, Quaternion rot, Transform parent)

// 从对象池获取一个声音实例对象或者创建一个声音实例对象
// 如果实例对象是null，就返回null
// 获取该对象的声音组件AudioSource, 并调用 play()
// 启动检测声音停止播放协程，如果声音停止播放了就调用Despawn。设置该对象为闲置状态
public AudioSource Spawn(AudioSource prefab, Vector3 pos, Quaternion rot)
public AudioSource Spawn(AudioSource prefab)
public AudioSource Spawn(AudioSource prefab, Transform parent)
public AudioSource Spawn(AudioSource prefab, Vector3 pos, Quaternion rot, Transform parent)

// 从对象池获取一个粒子特效实例对象或者创建一个粒子实例对象
// 如果实例对象是null，就返回null
// 启动检测粒子生命结束协程，如果粒子gameObject没激活就取消检测。emitter.IsAlive(true)==false时就调用Despawn。设置该对象为闲置状态
// 如果检测时间超过maxParticleDespawnTime就抛出一个警告
public ParticleSystem Spawn(ParticleSystem prefab, Vector3 pos, Quaternion rot)
public ParticleSystem Spawn(ParticleSystem prefab, Vector3 pos, Quaternion rot, Transform parent)
public ParticleEmitter Spawn(ParticleEmitter prefab, Vector3 pos, Quaternion rot)
public ParticleEmitter Spawn(ParticleEmitter prefab, Vector3 pos, Quaternion rot, string colorPropertyName, Color color)

// 将一个实例对象设置为闲置状态
public void Despawn(Transform instance)
public void Despawn(Transform instance, Transform parent)

// 延迟将一个实例对象设置为闲置状态
public void Despawn(Transform instance, float seconds)
public void Despawn(Transform instance, float seconds, Transform parent)
```

ObjectPool<T>属性

```
/ 缓存池这个Prefab的预加载数量。意思为一开始加载的数量!
public int preloadAmount = 1;

// 如果勾选表示缓存池所有的gameobject可以“异步”加载。
public bool preloadTime = false;

// 每几帧加载一个。
public int preloadFrames = 2;

// 延迟多久开始加载。
public float preloadDelay = 0;



// 是否开启对象实例化的限制功能。
public bool limitInstances = false;

// 限制实例化Prefab的数量，也就是限制缓冲池的数量，它和上面的preloadAmount是有冲突的，如果同时开启则以limitAmout为准。
public int limitAmount = 100;

// 如果我们限制了缓存池里面只能有10个Prefab，如果不勾选它，那么你拿第11个的时候就会返回null。如果勾选它在取第11个的时候他会返回给你前10个里最不常用的那个。
public bool limitFIFO = false;



// 是否开启缓存池智能自动清理模式。
public bool cullDespawned = false;

// 缓存池自动清理，但是始终保留几个对象不清理。
public int cullAbove = 50;

// 每过多久执行一遍自动清理，单位是秒。从上一次清理过后开始计时
public int cullDelay = 60;

// 每次自动清理几个游戏对象。
public int cullMaxPerPass = 5;



// 是否打印日志信息
public bool logMessages = false

// 强制关闭打印日志
private bool forceLoggingSilent = false;
```

ObjectPool<T>方法

```
// 构造方法
// 设置 prefabGO = prefab.gameObject;
// 新建 _spawned = new List<T>();
// 新建 _despawned = new List<T>();
internal void inspectorInstanceConstructor()

// 析构方法
// 当SpawnPool.OnDestroy时调用
// 销毁_spawned,_despawned两个列表里的对象, 并清空两个列表
// 设置 prefab,prefabGO,spawnPool为null
internal void SelfDestruct()

// 缓存自动清理是否启动了，用来检测是否启动清理
private bool cullingActive = false;

// 正在被使用的对象列表
internal List<T> _spawned = new List<T>();
public List<T> spawned { get { return new List<T>(this._spawned); } }
// 没有被使用的闲置对象列表
internal List<T> _despawned = new List<T>();
public List<T> despawned { get { return new List<T>(this._despawned); } }
// 对象的总数 = 使用的数量 + 闲置的数量
public int totalCount

// 已经预加载
// 在PreloadInstances（）方法中用到，用来判断是否需要执行预加载
private bool _preloaded = false;
internal bool preloaded

// 设置对象为闲置
// 将对象xfrom从 _spawned转到_despawned
// sendEventMessage=true时会对xfrom对象广播OnDespawned消息
// 设置xfrom.gameObejct.SetActive(false)
// 检测是否需要启动自动清理
internal bool DespawnInstance(T xform)
internal bool DespawnInstance(T xform, bool sendEventMessage)

// 清理闲置对象协程
internal IEnumerator CullDespawned()
```

```
        // 获取一个实例对象
        // 如果限制了实例对象数量，且正在使用的对象数量大于限制的数量，且开启了limitFIFO，那么就把_spawned[0]设置为闲置状态
        // 如果闲置列表里有对象，就从限制列表里取出第一个对象；并把这个对象放到_spawned列表
        // 如果限制列表没有对象，就新建一个对象
        internal T SpawnInstance(Vector3 pos, Quaternion rot)

        // 创建一个实例对象
        // 如果限制了实例对象数量，且对象总数大于限制的数量。就返回一个空对象 return null
        // 否则 实例化一个对象，并检查SpawnPool的设置，对该对象进行设置。最后返回该创建的对象
        public T SpawnNew() { return this.SpawnNew(Vector3.zero, Quaternion.identity); }
        public T SpawnNew(Vector3 pos, Quaternion rot)

        // 递归设置xform和他childs的layer
        private void SetRecursively(T xform, int layer)
        // 将以个势力对象添加到缓存池。 despawn=true时就加到_despwan闲置列表，并设置gameObject.active=false；否则就加到_spawn正在被使用列表
        internal void AddUnpooled(T inst, bool despawn)

        // 预实例化对象
        // 如果preloaded=true。就表示已经预实例过了，终止执行
        // 如果开启实例对象数量限制limitInstances=true且预实例数量大于限制数量preloadAmount > limitAmount。那边就设置预实例数量等于限制数量preloadAmount = limitAmount;
        // 如果开启异步预实例对象，就开启延迟异步预实例化对象协程。如果每帧预实例化数量大于预实例化数量preloadFrames > preloadAmount，那么就设置每帧预实例化数量等于预实例化数量preloadFrames preloadAmount
        internal void PreloadInstances()
        // 延迟异步预实例化对象
        private IEnumerator PreloadOverTime()
        // 检查是否包含了该对象。spawned、despawned从这两个列表里检查
        public bool Contains(T T)
        // 再实例对象的名字后面加上（totalCount + 1 ）
        private void nameInstance(T instance)
```

ObjectPool<T>扩展方法

```
/** 实例化一个对象 */
virtual protected T Instantiate(params object[] arg)

/** 调用对象方法--实例对象重设参数 */
virtual protected void ItemSetArg(T instance, params object[] args)

/** 给实例对象重命名 */
virtual protected void nameInstance(T instance)


/** 调用对象方法--销毁 */
 virtual protected void ItemDestruct(T instance)

 /** 调用对象方法--设置为闲置状态消息 */
virtual protected void ItemOnDespawned(T instance)

 /** 调用对象方法--设置为使用状态消息 */
virtual internal void ItemOnSpawned(T instance)


 /** 调用对象方法--设置是否激活 */
virtual protected void ItemSetActive(T instance, bool value)
```

IPoolItem

```
public interface IPoolItem
   {
        /** 对象池的名称描述 */
        string PName{ get; set;}

        /** 销毁 */
        void PDestruct();

        /** 对象池设置--设置为使用状态消息 */
        void POnSpawned<T>(ObjectPool<T> pool);

        /** 对象池设置--该对象为闲置状态 */
        void POnDespawned<T>(ObjectPool<T> pool);

        /** 对象池设置--该对象是否激活 */
        void PSetActive(bool value);

        /** 对象池设置--该对象重设参数 */
        void PSetArg(params object[] args);

    }
```

实例—Prefab

```
public Transform prefab;
public PrefabPool pool = new PrefabPool();
public List<Transform> list = new List<Transform>();

public IEnumerator TestCache()
{

    pool.prefab = prefab;

    // 缓存池这个Prefab的预加载数量。意思为一开始加载的数量!
    pool.preloadAmount = 3;
    // 如果勾选表示缓存池所有的gameobject可以“异步”加载。
    pool.preloadTime = true;
    // 每几帧加载一个。
    pool.preloadFrames = 2;
    // 延迟多久开始加载。
    pool.preloadDelay = 0;
    // 是否打印日志信息
    pool.logMessages = true;


    PoolManager.groups.common.CreatePool(pool);

    yield return new WaitForSeconds(5);
    for(int i = 0; i < 20; i ++)
    {

        for(int j = 0; j < 10; j ++)
        {
            Transform item = PoolManager.groups.common.Spawn(prefab);
            item.position = Vector3.forward * (j + 0.5f);
            list.Add(item);
            Debug.LogFormat("[Spawn] { 0}, { 1}" , j, item);
            Debug.Log(pool);                    current =item;
            yield return new WaitForSeconds(1);
        }
        yield return new WaitForSeconds(5);


        for(int j = list.Count - 1; j >= 0; j --)
        {
            Transform item = list[j];
            PoolManager.groups.common.Despawn(item);
            Debug.Log(pool);
            yield return new WaitForSeconds(1);
        }
        list.Clear();

        yield return new WaitForSeconds(2);
    }
}
```

实例—ScriptableObject

```
 public class UnitData : ScriptableObject
    {
        private static int ID = 0;
        public int id;

        public UnitData()
        {
            id = ID ++;
        }


        public override string ToString()
        {
            return string.Format("[UnitData] id={ 0}", id);
        }
    }

ScriptableObjectPool<UnitData> unitDataPool = new ScriptableObjectPool<UnitData>();
 public IEnumerator TestCull()
        {

            // 缓存池这个Prefab的预加载数量。意思为一开始加载的数量!
            unitDataPool.preloadAmount = 3;
            // 如果勾选表示缓存池所有的gameobject可以“异步”加载。
            unitDataPool.preloadTime = true;
            // 每几帧加载一个。
            unitDataPool.preloadFrames = 2;
            // 延迟多久开始加载。
            unitDataPool.preloadDelay = 0;


            // 是否开启对象实例化的限制功能。
            unitDataPool.limitInstances = false;
            // 限制实例化Prefab的数量，也就是限制缓冲池的数量，它和上面的preloadAmount是有冲突的，如果同时开启则以limitAmout为准。
            unitDataPool.limitAmount = 5;
            // 如果我们限制了缓存池里面只能有10个Prefab，如果不勾选它，那么你拿第11个的时候就会返回null。如果勾选它在取第11个的时候他会返回给你前10个里最不常用的那个。
            unitDataPool.limitFIFO = true;

            // 是否开启缓存池智能自动清理模式
            unitDataPool.cullDespawned = true;
            // 缓存池自动清理，但是始终保留几个对象不清理。
            unitDataPool.cullAbove = 5;
            // 每过多久执行一遍自动清理，单位是秒。从上一次清理过后开始计时
            unitDataPool.cullDelay = 2;
            // 每次自动清理几个游戏对象。
            unitDataPool.cullMaxPerPass = 2;

            // 是否打印日志信息
            unitDataPool.logMessages = true;


            PoolManager.groups.common.CreatePool<UnitData>(unitDataPool);
            status = "Init";

            yield return new WaitForSeconds(5);
            Debug.Log("-----Spawn----");
            status = "Spawn ";
            for(int j = 0; j < 10; j ++)
            {
                UnitData unitData = PoolManager.groups.common.Spawn<UnitData>();
                unitDatas.Add(unitData);
                Debug.LogFormat("[Spawn] { 0}, { 1}" , j, unitData);
                Debug.Log(unitDataPool);
                status = "Spawn " + j;
                current =unitData != null ?  unitData.ToString() : "null";
                yield return new WaitForSeconds(1);
            }

            yield return new WaitForSeconds(2);
            Debug.Log("-----Despawn----");
            for(int j = unitDatas.Count - 1; j >= 0; j --)
            {
                UnitData unitData = unitDatas[j];
                PoolManager.groups.common.Despawn<UnitData>(unitData);
                Debug.Log(unitDataPool);
                status = "Despawn " + j;
                current =unitData != null ?  unitData.ToString() : "null";
                yield return new WaitForSeconds(1);
            }
}
```