

MaNGos Zero——Pool 分析报告
赵睿 2018K8009909001

一. 简介

本文是中国科学院大学面向对象课程的源码分析报告，主要分析了 MaNGos Zero 的 PoolManager 模块，也就是怪兽和可交互物品的管理。

二. 关键模块功能分析与建模

2.1 从游戏本身说起

进行源码分析之初，我认为通过阅读代码就可以对 PoolManager 模块有足够的了解和认识。但事实上，由于之前并没有接触过这款游戏，在阅读代码的过程中不同的类、结构体、变量与函数之间错综复杂的关系非常难以分析，于是我便先跳出了代码细节，对这款游戏，尤其是我要分析的部分，做了一个宏观的了解。

首先对代码中经常出现的几个名词进行一个解释：

Gameobject：游戏中的物品。

Creature：本意是生物，这里可以理解为游戏中的怪兽。

Pool：pool 可以理解为联合体，它将在代码中多次出现，不同的定义则表示了多个不同的联合体。（联合体中可能有怪兽、可能有物品）（祖孙联合体联系起了所有的东西）

Map：与它的本意相似，就是这个虚拟世界的地图。

Spawn：它的本意是生成，在游戏中意思与本意相似，就是让怪兽或物品进入到游戏世界。

Event：游戏中的事件。（这个事件可能关联着怪兽、物品、联合体）

至此，结合这些基础的了解，我们可以开始对源码进行了。这部分代码的功能比较琐碎，笼统而言，就是对怪兽、物品、联合体进行的一些操作，这些操作包括但不限于初始化、加载、检查、更新、出入游戏世界，但这部分代码本身并没有实现一个明确的、统一的功能。因此，我想从其中一个具体的功能入手来进行分析。

2.2 从一个功能入手

我选择了初始化这个功能进行分析。这个功能看似简单，但是调用关系复杂，涉及到了代码中出现的全部的类，由它入手进行分析，有助于我们对代码有一个更清楚的认知。

对联合体初始化的方法定义在 PoolManager 类中，我们可以叫它联合体管理器。对 PoolManager 类进行如下建模。

| | |
|----|---|
| 类 | 联合体管理器（PoolManager） |
| 方法 | <code>void Initialize(MapPersistentState* state)</code> 初始化联合体 <code>void InitSpawnPool(MapPersistentState& mapState, uint16 pool_id)</code> 初始化-生成联合体 <code>void SpawnPool(MapPersistentState& mapState, uint16 pool_id, bool instantly)</code> 生成联合体 <code>void SpawnPoolGroup(MapPersistentState& mapState, uint16 pool_id, uint32 db_guid_or_pool_id, bool instantly)</code> |
| 属性 | <code>mPoolTemplat</code> : PoolTemplateDataMap 联合体模板 <code>mPoolCreatureGroups</code> : PoolGroupCreatureMap 怪兽联合体 <code>mPoolGameobjectGroups</code> : PoolGroupGameObjectMap 物品联合体 <code>mPoolPoolGroups</code> : PoolGroupPoolMap 锚点池联合体 |

在联合体管理器中，有初始化联合体的方法，其定义代码如下。初始化联合体时的传递参数为地图的状态。遍历联合体模板 `mPoolTemplat`，如果生成开始于以 `pool_entry` 标记的这个联合体，即 `AutoSpawn` 返回 1，那么调用 `InitSpawnPool` 方法。

```

void PoolManager::Initialize(MapPersistentState* state)
{
    for (uint16 pool_entry = 0; pool_entry < mPoolTemplate.size(); ++pool_entry)
        if (mPoolTemplate[pool_entry].AutoSpawn)
        {
            InitSpawnPool(*state, pool_entry);
        }
}

```

InitSpawnPool 方法同样属于联合体管理器。它的定义代码如下。初始化-生成联合体 (InitSpawnPool) 时传递参数为地图状态和联合体的 id，并据此判断这个联合体是否可以进入地图，如果可以，调用 SpawnPool 方法。

```

void PoolManager::InitSpawnPool(MapPersistentState& mapState, uint16 pool_id)
{
    if (mPoolTemplate[pool_id].CanBeSpawnedAtMap(mapState.GetMapEntry()))
    {
        SpawnPool(mapState, pool_id, true);
    }
}

```

SpawnPool 方法同样定义在联合体管理器中，代码如下。生成联合体时传递的参数增加了一个布尔变量，表示是否立即生成，在本例中为 1。这个函数的主要作用是要生成的联合体分类（怪兽联合体、物品联合体、锚点池联合体），并根据分类调用不同的函数。

```

void PoolManager::SpawnPool(MapPersistentState& mapState, uint16 pool_id, bool instantly)
{
    SpawnPoolGroup<Pool>(mapState, pool_id, 0, instantly);
    SpawnPoolGroup<GameObject>(mapState, pool_id, 0, instantly);
    SpawnPoolGroup<Creature>(mapState, pool_id, 0, instantly);
}

```

以生成怪兽联合体为例继续向下分析。生成怪兽联合体需要用 SpawnPoolGroup<Creature>方法。这一方法同样定义在联合体管理器中，代码如下。生成怪兽联合体时传递的参数在之前的基础上增加了怪兽的 id (db_guid)。检查 mPoolCreatureGroups 容器中 pool_id 位置的怪兽联合体，然后调用 SpawnObject 方法。

```

void PoolManager::SpawnPoolGroup<Creature>(MapPersistentState& mapState, uint16 pool_id, uint32 db_guid, bool instantly)
{
    if (!mPoolCreatureGroups[pool_id].isEmpty())
    {
        mPoolCreatureGroups[pool_id].SpawnObject(mapState, mPoolTemplate[pool_id].MaxLimit, db_guid, instantly);
    }
}

```

在经过一系列条件判断、分类后，我们终于进入到了核心的生成联合体的流程。**SpawnObject** 方法虽然从字面意思上看是生成一个怪兽（也可以是物品，此处接上文以怪兽为例），但事实上是生成一个联合体中的所有怪兽。而 **Spawn10Object** 则是生成一个怪兽。**Spawn10Object** 与 **SpawnObject** 方法都定义在 **PoolGroup** 结构体中。对其建模如下。

| | |
|----|--|
| 类 | PoolGroup |
| 方法 | void SpawnObject (MapPersistentState& mapState, uint32 limit, uint32 triggerFrom, bool instantly)生成怪兽联合体 void Spawn10Object (MapPersistentState& mapState, PoolObject* obj, bool instantly)生成怪兽 |
| 属性 | EqualChanced : PoolObjectList ExplicitlyChanced : PoolObjectList poolId : uint32 |

取 **SpawnObject** 主要功能部分进行分析，代码如下。从图的状态中获得已经生成的联合体的信息，赋值给一个类指针 **spawns**。用联合体中能容纳的怪兽数减去已生成的怪兽数，得到联合体还能生成的怪兽数。随机取出一个怪兽，把这个怪兽的信息加到联合体的信息里，把这个怪兽放到地图里。这里需要用到 **Spawn10Object** 方法。**Spawn10Object** 实现根据怪兽的信息把怪兽从数据库中加载出来，并放到地图上。

```
void PoolGroup<T>::SpawnObject(MapPersistentState& mapState, uint32 limit, uint32 triggerFrom, bool instantly)
{
    SpawnedPoolData& spawns = mapState.GetSpawnedPoolData();
    uint32 lastDespawnd = 0;
    int count = limit - spawns.GetSpawnedObjects(poolId);
    if (triggerFrom)
        .....
    for (int i = 0; i < count; ++i)
    {
        PoolObject* obj = RollOne(spawns, triggerFrom);
        if (!obj)
            {continue;}
        .....
        spawns.AddSpawn<T>(obj->guid, poolId);
        Spawn10Object(mapState, obj, instantly);
        if (triggerFrom)
            .....
    }
}
```

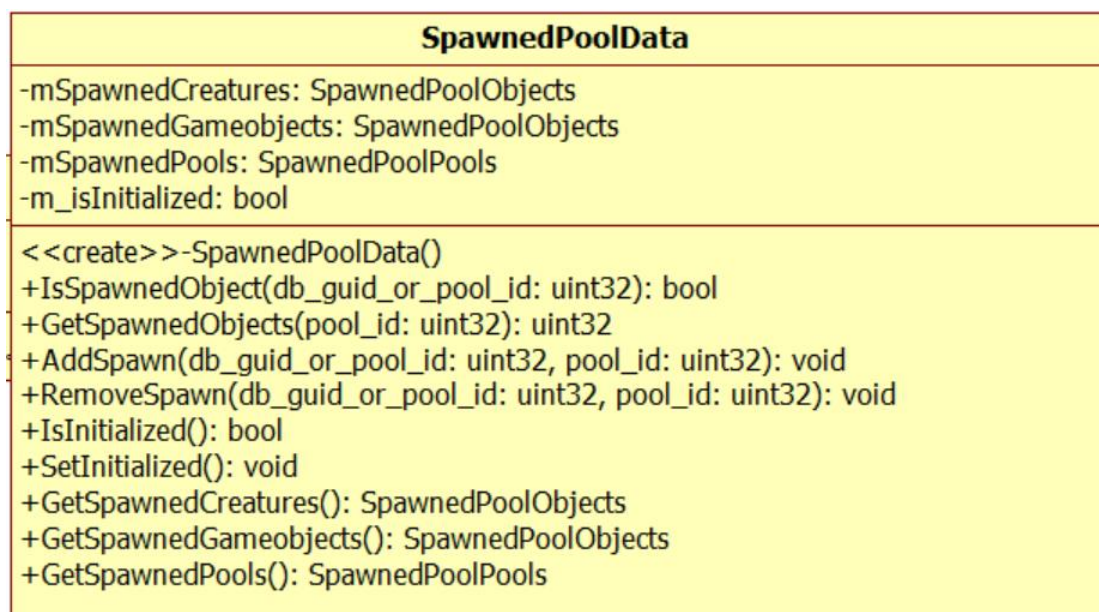
至此，我们已经基本了解了初始化联合体这一功能是如何实现的。这里要插一句题外话，在上面关于 **SpawnObject** 方法的分析中，我们用到了“信息”一词，这在代码中表现为“id”，引出了 **SpawnedPoolData** 类，可以叫它已生成的联合体的信息。这个类中的方法主要改变联合体、物品、怪兽的 **id**，而不改变物品、怪兽、联合体本身，因此这些方法往往要与另外两个类中的方法一同使用。

2.3 类图与模块主要功能

由于这部分代码没有一个统一的功能，因此我将按类分别具体阐述这个模块的功能。

2.3.1 SpawnedPoolData 类

正如上文所述，SpawnedPoolData 这个类中主要提供的就是针对“id”的操作，具体见下面的 UML 类图。



具体而言，其功能有获得联合体的信息，包括已生成对象（物品、怪兽、瞄准池）的数量、是否正在生成、添加生成对象的信息、删除生成对象的信息等。

2.3.2 PoolGroup 类

PoolGroup 这个类的主要功能是对联合体中的对象（怪兽、物品、瞄准池）进行一些检查与操作。



具体而言，其功能有：将对象添加到对应的队列中、检查联合体中取出对象的概率是否正确、从联合体中随机取出一个对象、检查事件与对象是否对应、对联合体中的对象进行操作（包括生成、丢弃、重生）等。

2.3.3 PoolManager 类

PoolManager 这个类的功能可以分为三方面，一方面，它建立起联合体与地图的关系，另一方面，它调用 PoolGroup 中的方法实现对联合体的操作。

| PoolManager |
|---|
| <pre>#max_pool_id: uint16 #mPoolTemplate: PoolTemplateDataMap #mPoolCreatureGroups: PoolGroupCreatureMap #mPoolGameObjectGroups: PoolGroupGameObjectMap #mPoolPoolGroups: PoolGroupPoolMap #mCreatureSearchMap: SearchMap #mGameObjectSearchMap: SearchMap #mPoolSearchMap: SearchMap <<create>>-PoolManager() <<destroy>>-PoolManager() +LoadFromDB(): void +Initialize(state: MapPersistentState): void +SetExcludeObject(pool_id: uint16, db_guid_or_pool_id: uint32, state: bool): void +CheckPool(pool_id: uint16): bool +CheckEventLinkAndReport(pool_id: uint16, event_id: int16, creature2event: std::map<uint32, int16>, go2event: std::map<uint32, int16>): void +SpawnPool(mapState: MapPersistentState, pool_id: uint16, instantly: bool): void +DespawnPool(mapState: MapPersistentState, pool_id: uint16): void +UpdatePool(mapState: MapPersistentState, pool_id: uint16, db_guid_or_pool_id: uint32): void +SpawnPoolInMaps(pool_id: uint16, instantly: bool): void +DespawnPoolInMaps(pool_id: uint16): void +InitSpawnPool(mapState: MapPersistentState, pool_id: uint16): void +UpdatePoolInMaps(pool_id: uint16, db_guid_or_pool_id: uint32): void +SpawnPoolGroup(mapState: MapPersistentState, pool_id: uint16, db_guid_or_pool_id: uint32, instantly: bool): void</pre> |

具体而言，其功能有：从数据库中加载对象与联合体、初始化联合体、生成联合体、检查联合体概率、检查时间链接、更新联合体；把联合体放进地图、从地图上删除、更新地图上的联合体等。

三．核心流程设计分析

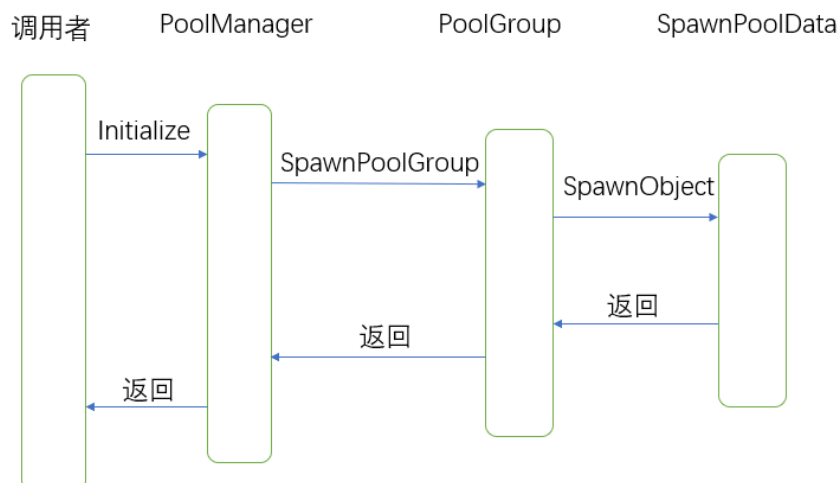
3.1 类间关系

在上文中我们已经了解了代码中对怪兽、物品、联合体进行管理所涉及的类以及这些类的主要功能，下面我们将分析这些类间的关系。

PoolManager 类是与 Map 相关联的类，它是怪兽、物品、联合体管理功能的主要入口，它依赖于 SpawnPoolGroup 接口，并组合到 PoolGroup 类上。SpawnPoolData 类用于对联合体的身份信息进行配置和处理。

3.2 执行流程

下面是初始化联合体的执行流程时序图，调用者调用 Initialize 方法，它进一步调用 SpawnPoolGroup、SpawnObject 方法来完成创建联合体中的怪兽、游戏物品、瞄准池，配置好联合体的身份信息，最后返回给调用者一个可以出现在游戏地图上的联合体：



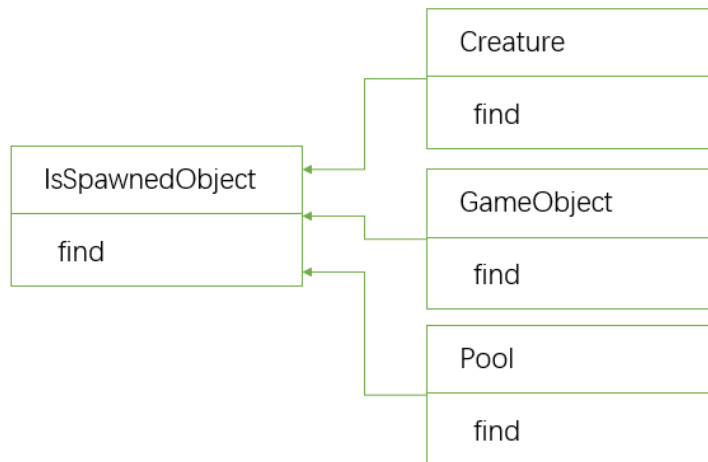
3.3 设计细节

在这一部分选取代码中体现的面向对象思想进行分析。

3.3.1 重载

本代码中通过函数模板实现重载，即用一个相同的方法名对不同对象实现不同方法。如

判断当前是否正在生成一个怪兽、物品或联合体的方法，采用相同的方法名：IsSpawnedObject，并对怪兽、物品或联合体分别进行定义。



代码如下：

```

template<>
bool SpawnedPoolData::IsSpawnedObject<Creature>(uint32 db_guid) const
{
    return mSpawnedCreatures.find(db_guid) != mSpawnedCreatures.end();
}
template<>
bool SpawnedPoolData::IsSpawnedObject<GameObject>(uint32 db_guid) const
{
    return mSpawnedGameobjects.find(db_guid) != mSpawnedGameobjects.end
();
}
template<>
bool SpawnedPoolData::IsSpawnedObject<Pool>(uint32 sub_pool_id) const
{
    return mSpawnedPools.find(sub_pool_id) != mSpawnedPools.end();
}

```

3.3.2 封装

以 `SpawnedPoolData` 类为例：

```

class SpawnedPoolData
{
public:
    SpawnedPoolData() : m_isInitialized(false) {}

    template<typename T>
    bool IsSpawnedObject(uint32 db_guid_or_pool_id) const;

    uint32 GetSpawnedObjects(uint32 pool_id) const;
}

```

```

template<typename T>
void AddSpawn(uint32 db_guid_or_pool_id, uint32 pool_id);

template<typename T>
void RemoveSpawn(uint32 db_guid_or_pool_id, uint32 pool_id);

bool IsInitialized() const { return m_isInitialized; }
void SetInitialized() { m_isInitialized = true; }

SpawnedPoolObjects const& GetSpawnedCreatures() const { return
mSpawnedCreatures; }
SpawnedPoolObjects const& GetSpawnedGameobjects() const { retur
n mSpawnedGameobjects; }
SpawnedPoolPools const& GetSpawnedPools() const { return mSpawne
dPools; }
private:
    SpawnedPoolObjects mSpawnedCreatures;
    SpawnedPoolObjects mSpawnedGameobjects;
    SpawnedPoolPools mSpawnedPools;
    bool m_isInitialized;
};

```

在这个类中，隐藏了怪兽、物品、瞄准池的 Id 属性，而公开了对它们的身份信息进行修改可使用的方法。

3.3.2 组合复用原则

在类间关系的分析中可以发现，当 PoolManager 类要使用 PoolGroup 类时、PoolGroup 类要使用 SpawnPoolData 类时，都是用了组合的方式，这符合优先使用组合而不是继承来达到复用的目的的组合复用原则。

3.3.3 单一职责原则

在本代码主要的三个类中，PoolManager 类负责对联合体进行操作、PoolGroup 类对联合体中的对象（怪兽、游戏物品、瞄准池）进行操作、SpawnPoolData 类对联合体的身份数据进行记录与操作，这符合一个类只负责做一件事的单一职责原则。

四．高级设计意图分析

这一部分我们主要进行对设计模式的讨论，主要分析了代码中所体现的设计模式。

4.1 观察者模式

观察者模式定义了对对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。观察者模式面向的需求是：A 对象（观察者）对 B 对象（被观察者）的某种变化高度敏感，需要在 B 变化的一瞬间做出反应。本代码中 PoolGroup 类与 PoolManager 类之间的关系符合观察者模式。PoolManager 类定义如下：

```

class PoolManager
{
public:
    PoolManager();
    ~PoolManager() {};

```

```

.....
    PoolGroup<Creature> const& GetPoolCreatures(uint16 pool_id) con
st { return mPoolCreatureGroups[pool_id]; }
    PoolGroup<GameObject> const& GetPoolGameObjects(uint16 pool_id)
const { return mPoolGameobjectGroups[pool_id]; }
    PoolGroup<Pool> const& GetPoolPools(uint16 pool_id) const { re
turn mPoolPoolGroups[pool_id]; }
protected:
    template<typename T>
    void SpawnPoolGroup(MapPersistentState& mapState, uint16 pool_i
d, uint32 db_guid_or_pool_id, bool instantly);

    uint16 max_pool_id;

    typedef std::vector<PoolGroup<Creature> > PoolGroupCreatureMa
p;
    typedef std::vector<PoolGroup<GameObject> > PoolGroupGameObject
Map;
    typedef std::vector<PoolGroup<Pool> > PoolGroupPoolMap;
    typedef std::pair<uint32, uint16> SearchPair;
    typedef std::map<uint32, uint16> SearchMap;

    PoolTemplateDataMap mPoolTemplate;
    PoolGroupCreatureMap mPoolCreatureGroups;
    PoolGroupGameObjectMap mPoolGameobjectGroups;
    PoolGroupPoolMap mPoolPoolGroups;

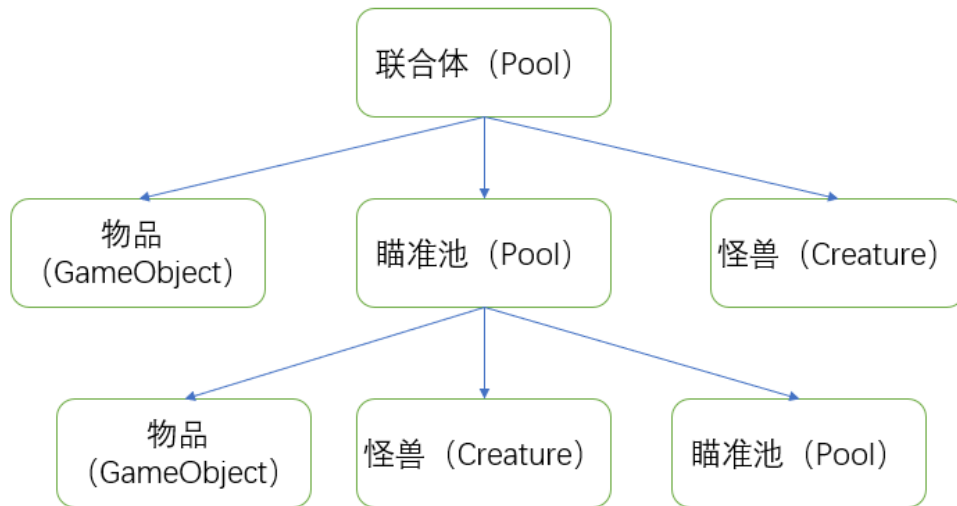
    // static maps DB low guid -> pool id
    SearchMap mCreatureSearchMap;
    SearchMap mGameobjectSearchMap;
    SearchMap mPoolSearchMap;
};

```

PoolManager 依赖于 PoolGroup，当 PoolGroup 发生改变时，意味着怪兽、物品、瞄准池发生改变，PoolManager 也必然发生更新。

4.2 组合模式

组合模式是将对象以树形结构组织起来，以达到“部分 – 整体”的层次结构，使得客户端对单个对象和组合对象的使用具有一致性。在本代码中，联合体、物品、怪兽、瞄准池符合如下的树形关系：



但本代码中并没有用组合模式实现这几个对象之间的关系，而是直接将物品、怪兽、瞄准池分别定义为 GameObject、Creature、Pool 类。在这一点上，我想或许作者有在本文所分析代码之外的考量，即这几个对象在本游戏的其他部分代码上存在一些不符合组合模式的地方，又或许作者确实在这一点上存在值得改进的地方。

五．小结

写到这里，也就意味着本学期的面向对象课程结束了，本次的报告仍有一些不足，尤其是关于设计模式的研讨，由于我本人对这部分的理解与学习不是很深入，因此对这部分代码设计模式的分析也并不是非常完美，这也鞭策着我在以后的日子里继续学习、思考。非常感谢老师与助教的阅读与指导。