# 网络同步原理分析

## 前言

UE同步是一块比较复杂而庞大的模块，里面设计到了很多设计思想，技巧，技术。我这里主要是从同步的流程分析，以同步的机制为讲解核心，给大家描述里面是怎么同步的，会大量涉及UE同步模块的底层代码，稍微涉及一点计算机网络底层（Socket相关）相关的知识。

**一． 基本概念与架构特点**

UE网络是一个相当复杂的模块，这篇文章主要是针对Actor同步，属性同步，RPC等大致的阐述一些流程以及关键的一些类。这里我尽可能将我的理解写下来。  
在UE里面有一些和同步相关的概念与类，这里逐个列举一下并做解释：

**基本网络通信：**

* NetDriver  
  网络驱动，实际上我们创建使用的是他的子类IPNetDriver，里面封装了基本的同步Actor的操作，初始化客户端与服务器的连接，建立属性记录表，处理RPC函数，创建Socket，构建并管理当前Connection信息，接收数据包等等基本操作。NetDriver与World一一对应，在一个游戏世界里面只存在一个NetDriver。UE里面默认的都是基于UDPSocket进行通信的。
* Connection  
  表示一个网络连接。服务器上，一个客户端到一个服务器的一个连接叫一个ClientConnection。在客户端上，一个服务器到一个客户端的连接叫一个ServerConnection。
* LocalPlayer  
  本地玩家，一个客户端的窗口ViewportClient对应一个LocalPlayer，Localplayer在各个地图切换时不会改变。
* Channel  
  数据通道，每一个通道只负责交换某一个特定类型特定实例的数据信息。ControlChannel：客户端服务器之间发送控制信息，主要是发送接收连接与断开的相关消息。在一个Connection中只会在初始化连接的时候创建一个该通道实例。  
  VoiceChannel：用于发送接收语音消息。在一个Connection中只会在初始化连接的时候创建一个该通道实例。  
  ActorChannel：处理Actor本身相关信息的同步，包括自身的同步以及子组件，属性的同步，RPC调用等。每个Connection连接里的每个同步的Actor都对应着一个ActorChannel实例。  
  常见的只有这3种：枚举里面还有FileChannel等类型，不过没有使用。
* PlayerController  
  玩家控制器，对应一个LocalPlayer，代替本地玩家控制游戏角色。同时对应一个Connection，记录了当前的连接信息，这和RPC以及条件属性复制都是密切相关的。另外，PlayerController记录他本身的ViewTarget（就是他控制额Character），通过与ViewTarget的距离（太远的Actor不会同步）来进行其他Actor的同步处理。
* World  
  游戏世界，任何游戏逻辑都是在World里面处理的，Actor的同步也受World控制，World知道哪些Actor应该同步，保存了网络通信的基础设施NetDriver。
* Actor  
  在世界存在的对象，没有坐标。UE4大部分的同步功能都是围绕Actor来实现的。
* Dormant  
  休眠，对于休眠的Actor不会进行网络同步

**底层通信：**

* Packet 从Socket读出来/输出的数据，一个Packet里面可能有多个Bunch数据或者Ack数据
* Bunch 一个Bunch里面主要记录了Channel信息，NGUID。同时包含其他的附属信息如是否是完整的Bunch，是否是可靠等，可以简单理解为一个从逻辑上层分发下来的同步数据包，该数据包的数据可能不完整，Bunch分为属性Bunch以及RPCBunch。继承自FNetBitWriter InBunch：从Channel接收的数据流串 ，UNetConnection::ReceivedPacket的时候创建 OutBunch：从Channel产生的数据流串，UActorChannel::ReplicateActor()的时候创建
* Ack Ack是与Bunch同级别概念的网络数据串，用于实现UDP的可靠数据传输
* FBitWriter 字节流书写器，可以临时写入比特数据用于传输，存储等，继承自FArchive
* FSocket 所有平台Socket的基类。 FSocketBSD：使用winSocket的Socket封装
* UPackageMap 生成与维护Object与NGUID的映射，负责Object的序列化。每一个Connection对应一个UPackageMap
* PacketHandler ：网络包预处理，比如加密，前向纠错，握手等。里面有一个或多个HandlerComponents来执行特殊的数据处理。目前内置的包括加密组件RSA，AES，以及必备的握手组件StatelessConnectHandlerComponent

（Packet与Bunch的区别：Bunch是Packet子集，Packet里面可能不包含Bunch信息，只包含Ack数据）

**属性同步相关：**

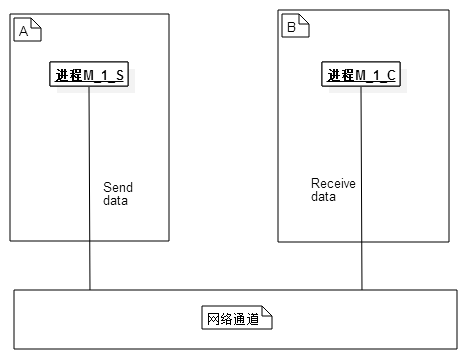
* FObjectReplicator  
  属性同步的执行器，每个Actorchannel对应一个FObjectReplicator，每一个FObjectReplicator对应一个对象实例。设置ActorChannel通道的时候会创建出来。
* FRepState  
  针对每个连接同步的历史数据，记录同步前用于比较的Object对象信息，存在于FObjectReplicator里面。
* FRepLayOut  
  同步的属性布局表，记录所有当前类需要同步的属性，每个类或者RPC函数有一个。
* FRepChangedPropertyTracker  
  属性变化轨迹记录，一般在同步Actor前创建，Actor销毁的时候删掉。
* FReplicationChangelistMgr  
  存放当前的Object对象，保存属性的变化历史记录

**架构特点：**

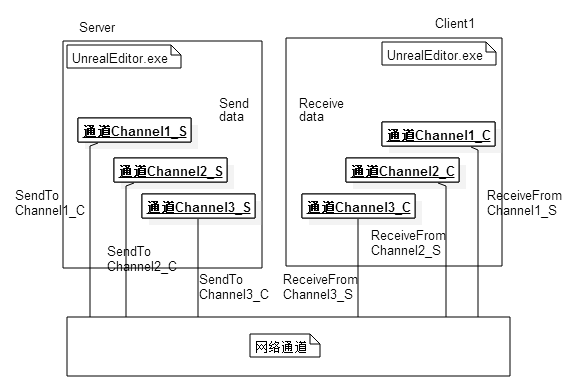
* 客户端服务器共用一套代码
* 服务器为游戏逻辑服务器，单个服务器为核心，多个客户端连接
* 默认通信协议为UDP（应用层实现数据可靠的UDP）
* 收发UDP数据包都在主线程（GameThread）执行

**二． 通信的基本流程**

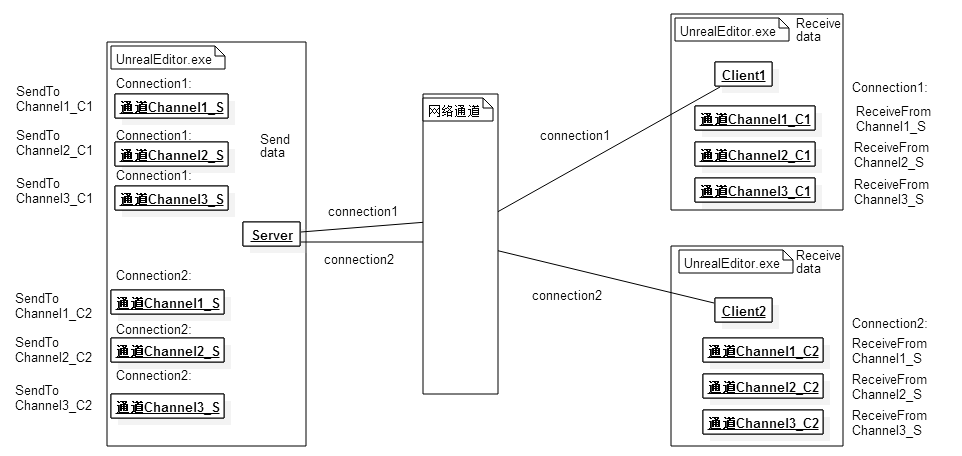
如果我们接触过网络通信，应该了解只要知道对方的IP地址以及端口号，服务器A上进程M\_1\_Server可以通过套接字向客户端B上的进程M\_1\_Client发送消息，大致的效果如下：

图2-1 远程进程通信图

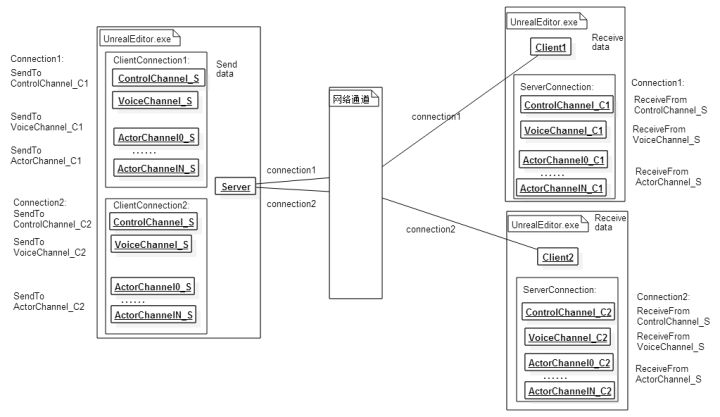
而对于UE4进程内部服务器Server与客户端Client1的通信，与上面的模型基本相似：

图2-2 UE4远程进程通信图

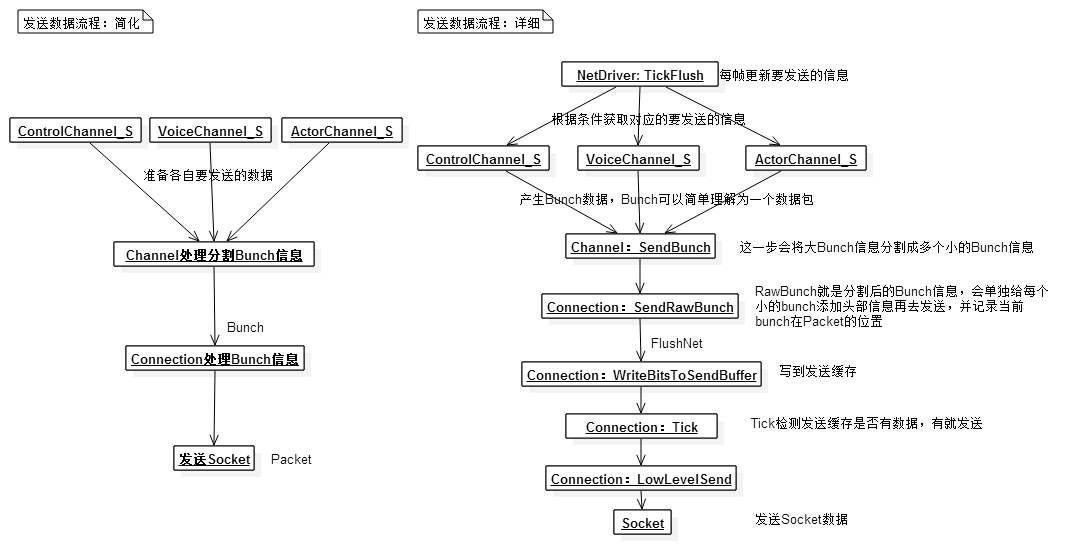
那这个里面的Channel是什么意思呢？简单理解起来就是一个通信轨道。为了实现规范与通信效率，我们的一个服务器针对某个对象定义了Channel通道，这个通道只与客户端对应的Channel通道互相发送与接收消息。这个过程抽象起来与TCP/UDP套接字的传输过程很像，套接字是在消息发送到进程前就进行处理，来控制客户端进程A只会接收到服务器对应进程A的消息，而这里是在UnrealEditor.exe进程里面处理，让通道1只接收到另一端通道1发送的消息。  
上面的只是针对一个服务器到客户端的传输流程，那么如果是多个客户端呢？

图2-3 Channel通信图

每一个客户端叫做一个Connection，如图，就是一个server连接到两个客户端的效果。对于每一个客户端，都会建立起一个Connection。在服务器上这个Connection叫做ClientConnection，对于客户端这个Connection叫做ServerConnection。每一个Channel都会归属于一个Connection，这样这个Channel才知道他对应的是哪个客户端上的对象。  
接下来我们继续细化，图中的Channel只标记了1,2,3，那么实际上都有哪些Channel？这些Channel对应的都是什么对象？其实，在第一部分的概念里我已经列举了常见的3中Channel，分别是ControlChannel，ActorChannel,以及VoiceChannel。一般来说，ControlChannel与VoiceChannel在游戏中只存在一个，而ActorChannel则对应每一个需要同步的Actor，所以我们再次细化上面的示意图：

图2-4 Connection下的Channel通信图

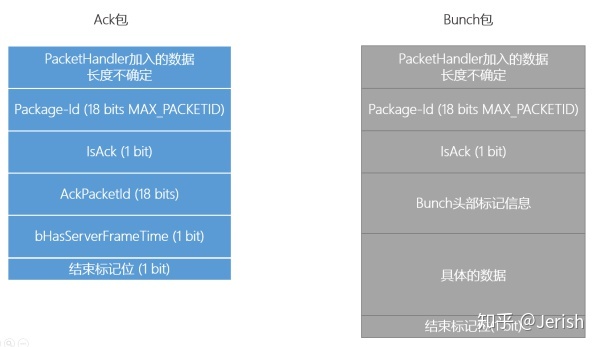
到这里我们基本上就了解了UE4的基本通信架构了，下面我们进一步分析网络传输数据的流程。首先我们要知道，UE4的数据通信是建立在UDP-Socket的基础上的，与其他的通信程序一样，我们需要对Socket的信息进行封装发送以及接收解析。流程如图所示：

图2-5 发送同步信息流程图图2-6 接收同步信息流程图

**1.数据包格式概述**

关于网络包的封装与解析，主要涉及到Bunch，RawBunch，Packet等概念，可以参考第一部分的基本概念去理解，最后一部分可靠数据传输还会讲到。每次通道最后发出去的包都成为一个Packet，Packet里面的信息是通过Bunch或者Ack塞进去的（也就是UNetConnection的Sendbuffer），里面可能有多个bunch与Ack，数量取决于你Flush的频率，默认情况下是在NetDriver Tick的时候或者buffer到达一定值的时候发送。

在Channel、Connection类里面有许多具体的函数通过处理Bunch/Packet实现了可靠传输，文章的最后一部分会做分析。

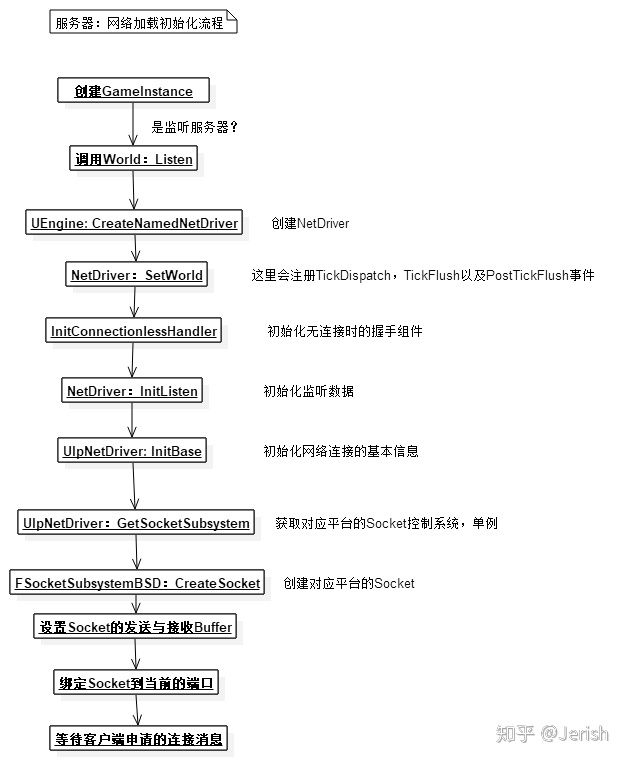


**三． 连接的建立**

前面的内容已经提到过，UE的网通通信是基于Channel的，而ControlChannel就是负责  
控制客户端与服务器建立连接的通道，所以客户端与服务器的连接信息都是通过UControlChannel执行NotifyControlMessage函数处理的。下面首先从服务器与客户端的网络模块初始化说起，然后描述二者连接建立的详细流程：

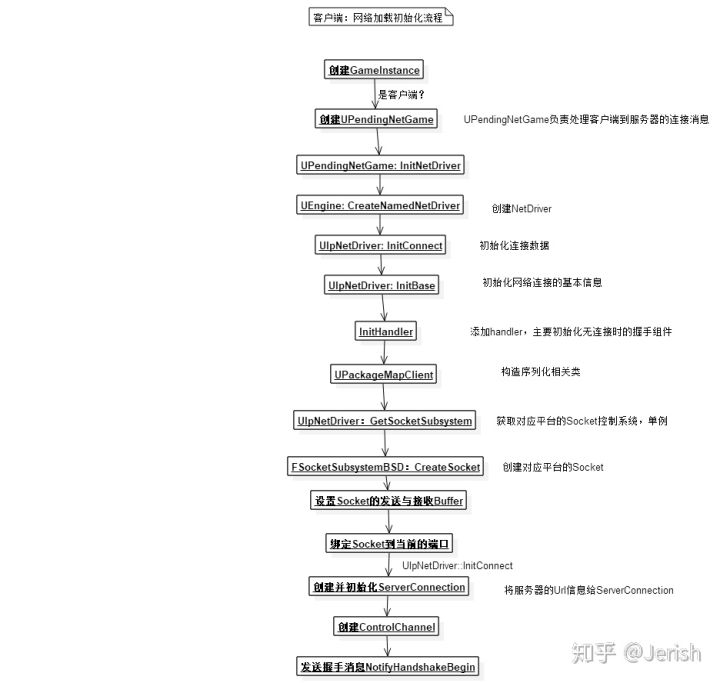
**1.服务器网络模块初始化流程**

从创建GameInstance开始，首先创建NetDriver来驱动网络初始化，进而根据平台创建对应的Socket，之后在World里面监听客户端的消息。

图3-1 服务器网络模块初始化流程图

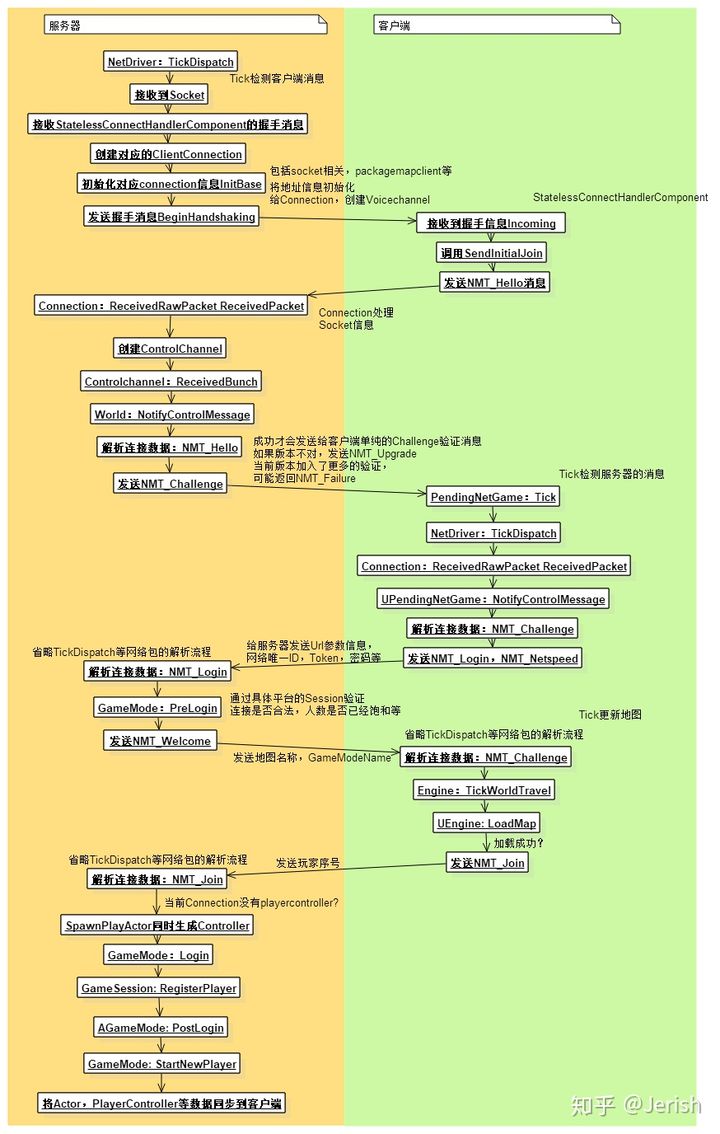
**2.客户端网络模块初始化流程**

客户端前面的初始化流程与服务器很相似，也是首先构建NetDriver，然后根据平台创建对应的Socket，同时他还会创建一个到服务器的ServerConnection。由于客户端没有World信息，所以要使用一个新的类来检测并处理连接信息，这个类就是UpendingNetGame。

图3-2 客户端网络模块初始化流程图

**3.服务器与客户端建立连接流程**

二者都完成初始化后，客户端就会开始发送一个Hello类型的ControlChannel消息给服务器（上面客户端初始化最后一步）。服务器接收到消息之后开始处理，然后会根据条件再给客户端发送对应的消息，如此来回处理几个回合，完成连接的建立，详细流程参考下图：  
（该流程是本地局域网的连接流程，与在线查找服务器列表并加入有差异）

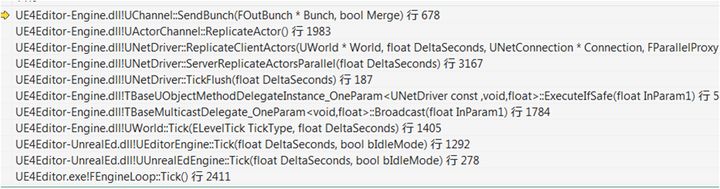
图3-3 客户端服务器连接建立流程图（已更新）

**四． Actor的同步细节**

Actor的同步可以说是UE4网络里面最大的一个模块了，里面包括属性同步，RPC调用等，这里为了方便我将他们拆成了3个部分来分别叙述。  
有了前面的描述，我们已经知道NetDiver负责整个网络的驱动，而ActorChannel就是专门用于Actor同步的通信通道。  
这里对Actor同步做一个比较细致的描述：服务器在NetDiver的TickFlush里面，每一帧都会去执行ServerReplicateActors来同步Actor的相关内容。在这里我们需要做以下处理:

1. 获取到所有连接到服务器的ClientConnections，首先获取引擎每帧可以同步的最大Connection的数量，超过这个限制的忽略。然后对每个Connection几乎都要进行下面所有的操作
2. 找到要同步的Actor，只有被放到World.NetworkActors里面的Actor才会被考虑，Actor在被Spawn时候就会添加到这个NetworkActors列表里面（新的版本里面已经把需要同的ACtor放到了NetDriver的NetworkObjects列表里面了）
3. 找到客户端玩家控制的角色ViewTarget（ViewTaget与摄像机绑定在一起），这个角色的位置是决定其他Actor是否同步的关键
4. 验证Actor，对于要销毁的以及所有权Role为ROLE\_NONE的Actor不会同步
5. 是否到达Actor同步时间，Actor的同步是有一定频率的，Actor身上有一个NetUpdateTime，每次同步前都会通过下面这个公式来计算下一次Actor的同步时间，如果没有到达这个时间就会放弃本次同步Actor->NetUpdateTime = World->TimeSeconds + FMath::SRand() \* ServerTickTime + 1.f/Actor->NetUpdateFrequency;
6. 如果这个Actor设置OnlyRelevantToOwner，那么就会放到一个特殊的列表里面OwnedConsiderList然后只同步给属于他的客户端。否则会把Actor放到ConsiderList里面
7. 对于休眠状态的Actor不会进行同步，对于要进入休眠状态的Actor也要特殊处理关闭同步通道
8. 查看当前的Actor是否有通道Channel，如果没有，还要看看Actor是否已经加在了场景，没有加载就跳过同步
9. 接第8个条件——没有Channel的情况下，还会执行Actor::IsNetRelevantFor判断是否网络相关，对于不可见的或者太远的Actor会返回false，不会同步
10. Actor的同步数量可能非常大，所以有必要对所有的Actor进行一个优先级的排列  
    处理完上面的逻辑后会对优先级表里的所有Actor进行排序
11. 排序后，如果连接没有加载此 actor 所在的关卡，则关闭通道（如果存在）并继续  
    每 1 秒钟调用一次 AActor::IsNetRelevantFor，确定 actor 是否与连接相关，如果不相关的时间达到 5 秒钟，则关闭通道  
    如果要同步的Actor没有ActorChannel就给其创建一个并绑定Actor，执行同步并更新NetUpdateTime = Actor->GetWorld()->TimeSeconds + 0.2f \* FMath::FRand();  
    如果此连接出现饱和剩下的 actor会根据连接相关时间判断是否在下一个时钟更新
12. 执行UActorChannel::ReplicateActor执行真正的Actor同步以及内部数据的同步，这里会将Actor（PackageMap->SerializeNewActor），Actor子对象以及其属性序列化（ReplicateProperties）封装到OutBunch并发送给客户端  
    （备注：我们当前版本下面的逻辑都是写在UNetDriver::ServerReplicateActors里面，4.12以后的UE4已经分别把Connection预处理，获取同步Actor列表，优先级处理等逻辑封装到单独的函数里了，详见ServerReplicateActors\_BuildConsiderlist, ServerReplicateActors\_PrioritizedActors, ServerReplicateActors\_ProsessPrioritizedActors等函数  
    优先级排序规则是什么？答案是按照是否有controller，距离以及是否在视野。通过FActorPriority构造代码可以定位到APawn::GetNetPriority,这里面会计算出当前Actor对应的优先级，优先级越高同步越靠前，是否有Controller的权重最大）  
    总之，大体上Actor同步的逻辑就是在TickFlush里面去执行ServerReplicateActors，然后进行前面说的那些处理。最后对每个Actor执行ActorChannel::ReplicateActor将Actor本身的信息，子对象的信息，属性信息封装到Bunch并进一步封装到发送缓存中，最后通过Socket发送出去。

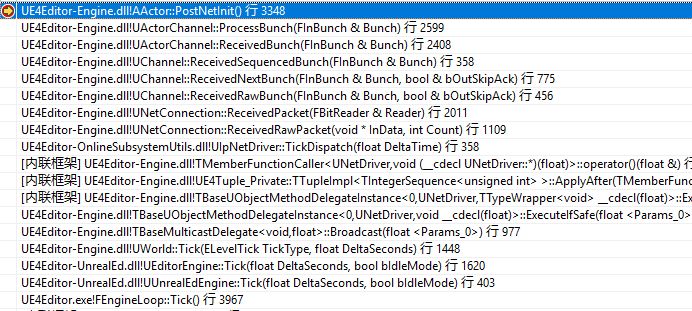
下面是服务器的同步Actor的发送Bunch堆栈：（代码修改过，与UE默认的有些不同）

图4-1 服务器同步Actor堆栈图

下面描述客户端是如何接收到服务器同步过来的Actor的。首先客户端TickDispatch检测服务器的消息，收到消息后通过Connection以及Channel进行解析，最后一步解析出完整数据的操作在UActorChannel::ProcessBunch执行，在这个函数里面：

1. 如果发现当前的ActorChannel对应的Actor为NULL，就对当前的Bunch进行反序列化Connection->PackageMap->SerializeNewActor(Bunch, this, NewChannelActor);解析出Actor的内容并执行PostInitializeComponents。如果Actor不为NULL，跳过这一步（参考下面图一堆栈）
2. 随后根据Bunch信息找到同步过来的属性值并对当前Actor对应的属性进行赋值
3. 最后执行PostNetInit调用Actor的BeginPlay。（参考下面堆栈）

下面截取了客户端接收到同步Actor并初始化的调用堆栈：

图4-2 客户端接收并序列化同步的Actor堆栈图图4-3 客户端初始化同步过来Actor堆栈图

从上面的描述来看，基本上我们可以很容易的分析出当前的Actor是否被同步，比如在UActorChannel::ReceivedBunch里面打个断点，看看当前通道里有没有你要的Actor就可以了。

**1.组件（子对象）同步**

组件（还有其他子对象）是挂在Actor上面的，所以组件的同步与Actor同步是紧密相连的，当一个Actor进行同步的时候会判断所有的子对象是否标记了Replicate，如果标记了，就对其以及其属性进行同步。

这些子对象同步方式（RPC等）也与Actor相差无几，实际上他们想要同步的话需要借助ActorChannel创建自己的FObjectReplicator以及属性同步的相关数据结构。简单来说，就是一个Actor身上的组件同步需要借用这个Actor的通道来进行。下面3段代码是服务器序列化子对象准备发送的逻辑：

//UActorChannel::ReplicateActor() DataChannel.cpp

// The Actor

WroteSomethingImportant |= ActorReplicator->ReplicateProperties( Bunch, RepFlags );

// 子对象的同步操作

WroteSomethingImportant |= Actor->ReplicateSubobjects(this, &Bunch, &RepFlags);

//ActorReplication.cpp

boolAActor::ReplicateSubobjects(UActorChannel \*Channel, FOutBunch \*Bunch, FReplicationFlags \*RepFlags)

{

check(Channel);

check(Bunch);

check(RepFlags);

bool WroteSomething = false;

for (int32 CompIdx =0; CompIdx < ReplicatedComponents.Num(); ++CompIdx )

{

UActorComponent \* ActorComp = ReplicatedComponents[CompIdx].Get();

//如果组件标记同步

if (ActorComp && ActorComp->GetIsReplicated())

{

WroteSomething |= ActorComp->ReplicateSubobjects(Channel, Bunch, RepFlags); // Lets the component add subobjects before replicating its own properties.检测组件否还有子组件

WroteSomething |= Channel->ReplicateSubobject(ActorComp, \*Bunch, \*RepFlags); // (this makes those subobjects 'supported', and from here on those objects may have reference replicated) 同步该组件

}

}

return WroteSomething;

}

//DataChannel.cpp

boolUActorChannel::ReplicateSubobject(UObject \*Obj, FOutBunch&Bunch, constFReplicationFlags&RepFlags)

{

if ( !Connection->Driver->GuidCache->SupportsObject( Obj ) )

{

FNetworkGUID NetGUID = Connection->Driver->GuidCache->AssignNewNetGUID\_Server(Obj ); //Make sure he gets a NetGUID so that he is now 'supported'

}

bool NewSubobject = false;

if (!ObjectHasReplicator(Obj))

{

Bunch.bReliable = true;

NewSubobject = true;

}

//组件的属性同步需要先在当前的ActorChannel里面创建新的FObjectReplicator

bool WroteSomething = FindOrCreateReplicator(Obj).Get().ReplicateProperties(Bunch, RepFlags);

if (NewSubobject && !WroteSomething)

{

......

}

return WroteSomething;

}

下面一段代码是客户端接收服务器同步过来的子对象逻辑：

// void UActorChannel::ProcessBunch( FInBunch & Bunch )DataChannel.cpp

// 该函数前面的代码主要是是进行反序列化当前Actor的相关操作

while ( !Bunch.AtEnd() && Connection != NULL&& Connection->State != USOCK\_Closed )

{

bool bObjectDeleted = false;

//当前通道的Actor以及反序列化成功，这里开始继续从Bunch里面寻找子对象进行反序列化

//如果当前Actor没有子组件，这里返回的就是Actor自身

......

TSharedRef<FObjectReplicator>& Replicator = FindOrCreateReplicator( RepObj );

bool bHasUnmapped = false;

// 找到当前子对象（或当前Actor）的Replicator以后，这里开始进行属性值的读取了

if ( !Replicator->ReceivedBunch( Bunch, RepFlags, bHasUnmapped ) )

{

......

}

......

}

前面Actor同步有提到，当从ActorChannel解析Bunch信息的时候就可以尝试对该数据流进行Actor的反序列化。从这段代码可以进一步看出，Actor反序列化之后会立刻开始判断Bunch里面是否存在其子对象，如果存在还会进一步读取子对象同步过来的属性值。如果没有子对象，就读取自身同步过来的属性。

关于子组件的反序列化还分为两种情况。要想理解这两种情况，还需要清楚两个概念——动态组件与静态组件。

对于静态组件：一旦一个Actor被标记为同步，那么这个Actor身上默认所挂载的组件也会随Actor一起同步到客户端（也需要序列化发送）。什么是默认挂载的组件?就是C++构造函数里面创建的默认组件或者在蓝图里面添加构建的组件。所以，这个过程与该组件是否标记为Replicate是没有关系的。

对于动态组件：就是我们在游戏运行的时候，服务器创建或者删除的组件。比如，当玩家走进一个洞穴时，给洞穴里面的火把生成一个粒子特效组件，然后同步到客户端上，当玩家离开的时候再删除这个组件，玩家的客户端上也随之删除这个组件。

对于动态组件，我们必须要设置他的Replicate属性为true，即通过函数 AActorComponent::SetIsReplicated(true)来操作。而对于静态组件，如果我们不想同步组件上面的属性，我们就没有必要设置Replicate属性。下面截取了函数ReadContentBlockHeader部分代码来区分这两种情况：

//静态组件，不需要客户端Spawn

FNetworkGUID NetGUID;

UObject \* SubObj = NULL;

Connection->PackageMap->SerializeObject(Bunch, UObject::StaticClass(), SubObj, &NetGUID );

//动态组件，需要在客户端Spawn出来

FNetworkGUID ClassNetGUID;

UObject \* SubObjClassObj = NULL;

Connection->PackageMap->SerializeObject(Bunch, UObject::StaticClass(), SubObjClassObj, &ClassNetGUID );

我们在这两段代码看到了FNetworkGUID的使用，因为这里涉及到UObject的引用（指针）同步。对于不同端的同一个对象，他们的内存地址肯定是不同的，那服务器上指向A的指针同步到客户端上如何也能正确的指向A呢？这就需要通过FNetworkGUID来解析。

**五． 属性同步细节**

**1.属性同步概述**

属性同步是一个很复杂的模块，我在另一个关于[UE4网络思考文章](https://zhuanlan.zhihu.com/p/34721113)里面讲解了属性同步相关的使用逻辑以及注意事项。这里我尽可能的分析一下属性同步的实现原理。有一点需要先提前说明一下，服务器同步的核心操作就是比较当前的同步属性是否发生变化，如果发生就将这个数据通过到客户端。如果是普通逻辑处理，我们完全可以保存当前对象的一个拷贝对象，然后每帧去比较这个拷贝与真实的对象是否发生变化。不过，由于同步数据量巨大，我们不可能给每个需要同步的对象都创建一个新的拷贝，而且这个逻辑如果暴露到逻辑层的话会使代码异常复杂难懂，所以这个操作要统一在底层处理。那么，UE4的基本思路就是获取当前同步对象的空间大小，然后保存到一个buffer里面，然后根据属性的OffSet给每个需要同步的属性初始化。这样，就保存了一份简单的“拷贝”用于之后的比较。当然，我们能这么做的前提是存在UE的Object对象反射系统。

（注意：虽然UE4的底层是可靠的UDP，但是不代表我们所有的网络数据都是可靠的。属性同步其实就并不完全可靠，除了一开始同步Actor的时候，属性同步是可靠的，但是后期的发送并不是，但是引擎内部有一定的机制可以保证客户端能收到服务器最后同步的属性消息，文章的最后一部分会谈到这些）

**下面开始进一步描述属性同步的基本思路：**我们给一个Actor类的同步属性A做上标记Replicates（先不考虑其他的宏），然后UClass会将所有需要同步的属性保存到ClassReps列表里面，这样我们就可以通过这个Actor的UClass获取这个Actor上所有需要同步的属性，当这个Actor实例化一个可以同步的对象并开始创建对应的同步通道时，我们就需要准备属性同步了。

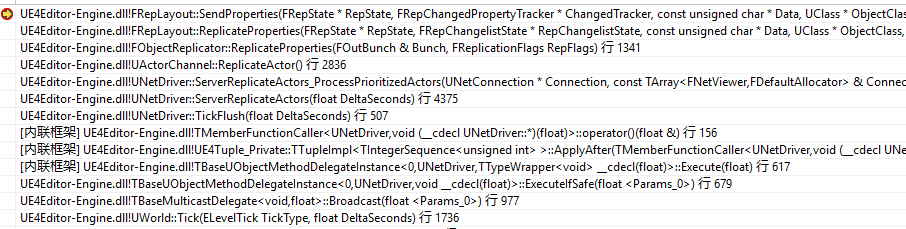
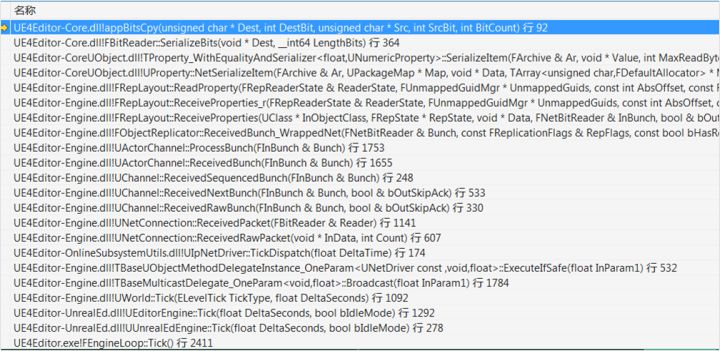
首先，我们要有一个同步属性列表来记录当前这个类有哪些属性需要同步**（FRepLayout，每个需要同步的网络对象【一个actor对象或者一个RPC对象】有一个，从UClass里面初始化）**；其次，我们需要针对每个对象保存一个缓存数据，来及时的与发生改变的Actor属性作比较，从而判断与上一次同步前是否发生变化**（FRepState，里面有一个Staticbuff来保存）**；然后，我们要有一个属性变化跟踪器记录所有发生改变同步属性的序号（可能是因为节省内存开销等原因所以不是保存这个属性），便于发送同步数据时处理**（FRepChangedPropertyTracker，对各个Connection可见，被各个Connection的Repstate保存一个共享指针，新版本被FRepChangelistState替换）**。最后，我们还需要针对每个连接的每个对象有一个控制前面这些数据的执行者**（FObjectReplicator）**。

这四个类就是我们属性同步的关键所在，在同步前我们需要对这些数据做好初始化工作，然后在真正同步的时候去判断与处理。

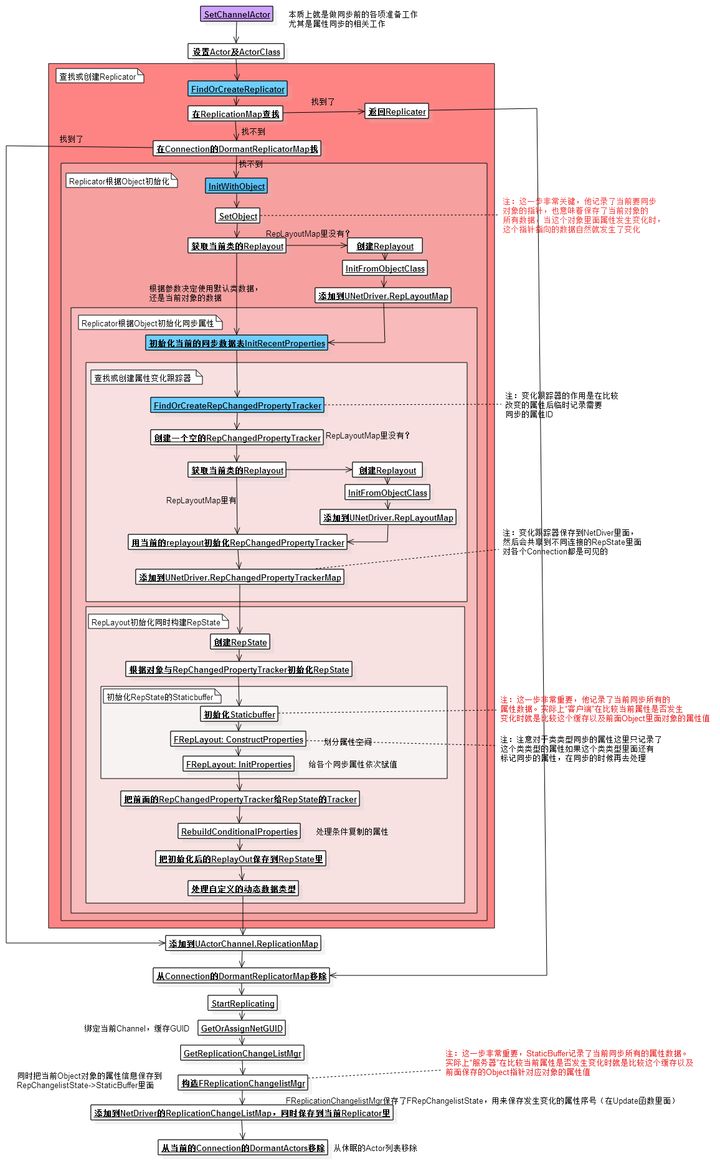
注：在4.12后的版本，新增了一个属性，FReplicationChangelistMgr。FReplicationChangelistMgr 里面保存了FRepChangelistState，FRepChangelistState属性可谓是兼顾FRepState以及FRepChangedPropertyTracker双重功能，他里面有一个Staticbuff来保存Object对象一个缓存数据，用来在服务器比较对象属性是否发生变化，同时又有一个FRepChangedHistory来记录所有发生过属性变化的历史记录[大小有限制]。然而，这不代表他能替代FRepState与FRepChangedPropertyTracker。目前，客户端在检测属性是否发生变化时使用的仍旧是RepState里面的Staticbuff。在处理条件属性复制时的判断使用的仍然是FRepChangedPropertyTracker

**2.重要数据的初始化流程**

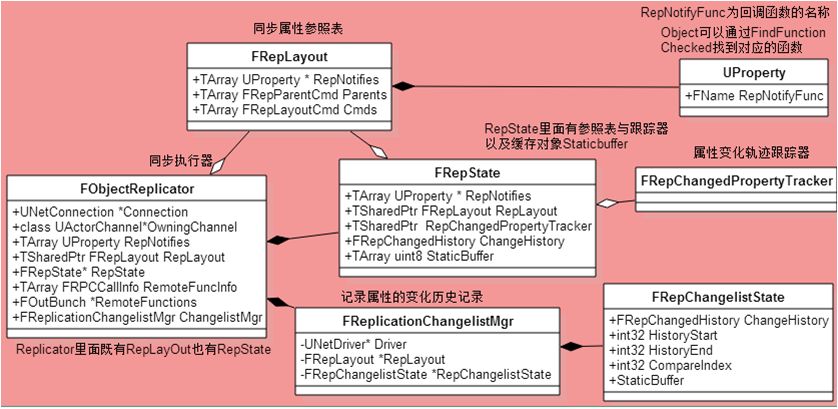
下面的两个图分别是属性同步的服务器发送堆栈以及客户端的接收堆栈。

图5-1服务器发送属性堆栈图图5-2客户端接收属性堆栈图

从发送堆栈中我们可以看到属性同步是在执行ReplicatActor的同时进行的，所以我们也可以猜到属性同步的准备工作应该与Actor的同步准备工作是密不可分的。前面Actor同步的讲解中我们已经知道，当Actor同步时如果发现当前的Actor没有对应的通道，就会给其创建一个通道并执行SetChannelActor。这个SetChannelActor所做的工作就是属性同步的关键所在，这个函数里面会对上面四个关键的类构造并做初始化，详细的内容参考下图：

图5-3 SetChannelActor流程解析图

图中详细的展示了几个关键数据的初始化，不过第一次看可能对这个几个类的关系有点晕，下面给大家简单画了一个类图。

图5-4属性同步相关类图

具体来说，每个ActorChannel在创建的时候会创建一个FObjectReplicator用来处理所有属性同步相关的操作，同时会把当前对应通道Actor的同步的属性记录在FRepLayOut的Parents数组里面（Parents记录了每个属性的UProperty，复制条件，在Object里面的偏移等）。

同时把这个RepLayOut存储到RepState里面，该RepState指针也会被存储到FObjectReplicator里面，RepState会申请一个缓存空间用来存放当前的Object对象（并不是完整对象，只包含同步属性，但是占用空间大小是一样的，用于客户端比较）

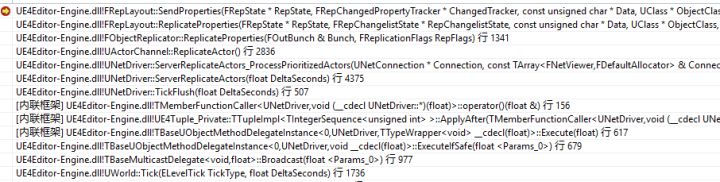
当然，FObjectReplicator还会保存一个指向FReplicationChangelistMgr的指针，指针对应对象里面的FRepChangelistState也申请一个缓存空间staticbuff用来存放当前的Object对象（用于服务器比较），同时还有一个ChangeHistory来保存属性的变化历史记录。

FRepChangedPropertyTracker在创建RepState的同时也被创建，然后通过FRepLayOut的Parents数量来初始化他的记录表的大小，主要记录对应的位置是否是条件复制属性，RepState里面保存一个指向他的指针。

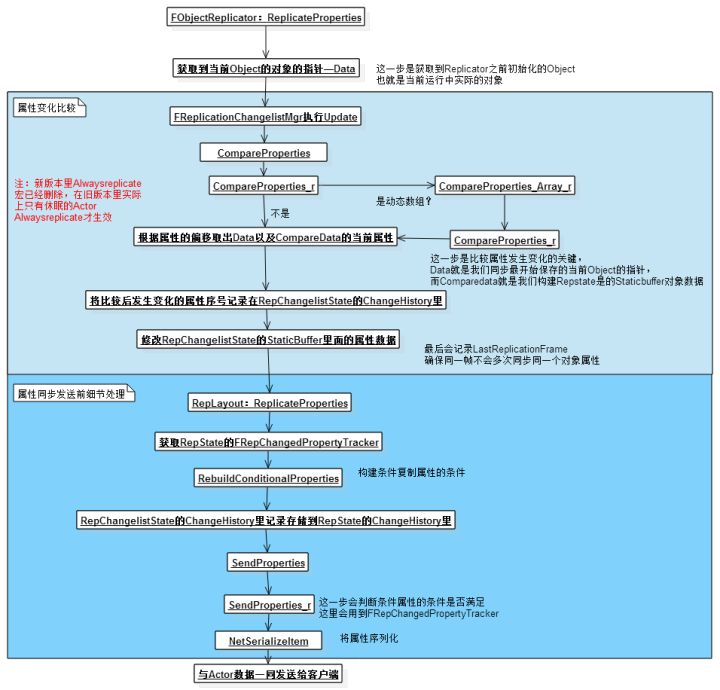
关于Parents属性与CMD属性：Replayout里面，数组parents示当前类所有的需要同步的属性，而数组cmd会将同步的复杂类型属性（包括数组、结构体、结构体数组但不包括类类型的指针）进一步展开放到这里面。比如ClassA里面有一个StructB属性，这个属性被标记同步，StructB属性会被放到parents里面。由于StructB里面有一个Int类型C属性以及D属性，那么C和D就会被放到Cmd数组里面。有关结构体的属性同步第5部分还有详细描述

**3.发送同步数据流程分析**

前面我们基本上已经做好了同步属性的基本工作，下面开始执行真正的同步流程。

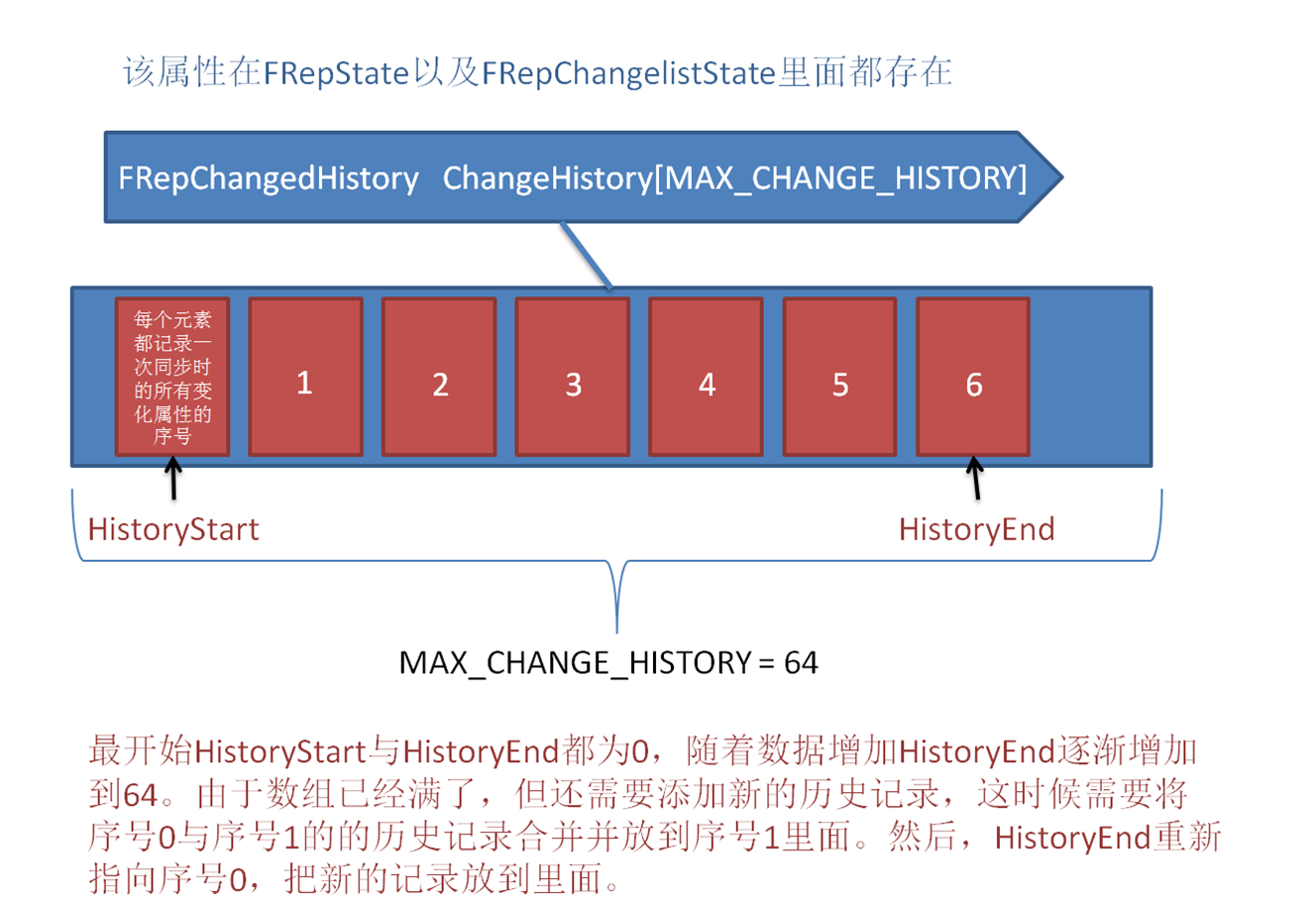
图5-5服务器发送属性堆栈图

再次拿出服务器同步属性的流程，我们可以看到属性同步是通过FObjectReplicator:: ReplicateProperties函数执行的，进一步执行RepLayout->ReplicateProperties。这里面比较重要的细节就是服务器是如何判断当前属性发生变化的，我们在前面设置通道Actor的时候给FObjectReplicator设置了一个Object指针，这个指针保存的就是当前同步的对象，而在初始化RepChangelistState的同时我们还创建了一个Staticbuffer，并且把buffer设置和当前Object的大小相同，对buffer取OffSet把对应的同步属性值添加到buffer里面。所以，我们真正比较的就是这两个对象，一般来说，staticbuffer在创建通道的同时自己就不会改变了，只有当与Object比较发现不同的时候，才会在发送前把属性值置为改变后的。这对于长期同步的Actor没什么问题，但是对于休眠的Actor就会出现问题了，因为每次删除通道并再次同步强制同步的时候这里面的staticbuff都是Object默认的属性值，那比较的时候就可能出现0不同步这样奇怪的现象了。真正比较两个属性是否相同的函数是PropertiesAreIdentical()，他是一个static函数。

图5-6 服务器同步属性流程图

**4. 属性变化历史记录**

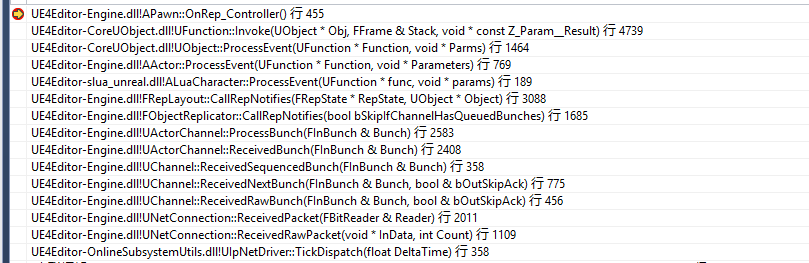
ChangeHistory属性在在FRepState以及FRepChangelistState里面都存在，不过每次同步前都是先更新FRepChangelistState里面的ChangeHistory，随后在发送前将FRepChangelistState的本次同步发生变化数据拷贝到FRepState的ChangeHistory本次即将发送的变化属性对应的数组元素里面。简单来说，就是FRepState的ChangeHistory一般只保存当前这一次同步发生变化的属性序号，而FRepChangelistState可以保存之前所有的变化的历史记录（更准确的说是最近的64次变化记录）。

图5-7

注意：合并的过程可能会出现同一个属性都发生变化的情况，不过由于里面只是记录了他是否发生过变化，所以合并是不会发生冲突的，只要有一个历史记录里面有变化就认为合并后的结果也有变化。

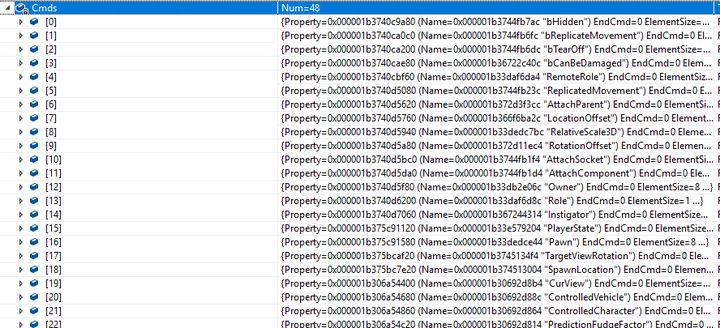
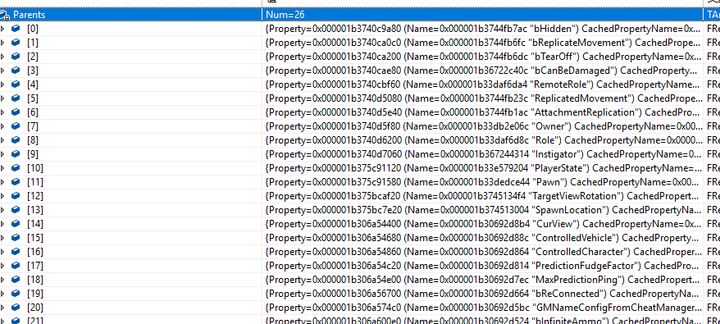
**5.属性回调函数执行**

虽然属性同步是由服务器执行的，但是FObjectReplicator，RepLayOut这些数据可并不是仅仅存在于服务器，客户端也是存在的，客户端也有Channel，也需要执行SetChannelACtor。不过这些数据在客户端上的作用可能就有一些变化，比如Staticbuffer，服务器是用它存储上次同步后的对象，然后与当前的Object比较看是否发生变化。在客户端上，他是用来临时存储当前同步前的对象，然后再把通过过来的属性复制给当前Object，Object再与Staticbuffer对象比较，看看属性是否发生变化，如果发生变化，就在Replicator的RepState里面添加一个函数回调通知RepNotifies。 在随后的ProcessBunch处理中，会执行RepLayout->CallRepNotifies( RepState, Object );处理所有的函数回调，所以我们也知道了为什么接收到的属性发生变化才会执行函数回调了。

图5-8 客户端属性回调堆栈图

**6.关于动态数组与结构体的同步**

结构体：UE里面UStruct类型的结构体与C++的Struct不一样，在反射系统中对应的是UScriptStruct，他本身可以被标记Replicated并且结构体内的数据默认都会被同步，而且如果里面有还子结构体的话也也会递归的进行同步。如果不想同步的话，需要在对应的属性标记NotReplicated，而且这个标记只对UStruct有效，对UClass无效。这一段的逻辑在FRepLayout::InitFromObjectClass处理，ReplayOut首先会读取Class里面所有的同步属性并逐一的放到FRepLayOut的数组Parents里面，这个Parents里面存放的就是当前类的继承树里面所有的同步属性。随后对Parents里面的属性进一步解析（FRepLayout:: InitFromProperty\_r），如果发现当前同步属性是数组或者是结构体就会对其进行递归展开，将数组的每一个元素/UStruct里面的每一个属性逐个放到FRepLayOut的Cmds数组里面，这个过程中如果遇到标记了NotReplicate的UStruct内部属性，就跳过。所以Cmds里面存放的就是对数组或者结构体进一步展开的详细属性。

图5-10 Cmds内部成员截图

**Struct ：**结构内的数据是不能标记Replicated的。如果你给Struct里面的属性标记Replicated，UHT在编译的时候就会提醒你编译失败”Struct members cannot be replicated”。这个提示多多少少会让人产生误解，实际上这个只是表明UStruct内部属性不能标记Replicated而已。最后，UE里面的UStruct不可以以成员指针的方式在类中声明。

**数组：**数组分为两种，静态数组与动态数组。静态数组的每一个元素都相当于一个单独的属性存放在Class的ClassReps里面，同步的时候也是会逐个添加到RepLayOut的Parents里面，参考上面的图5-9。UE里面的动态数组是TArray，他在网络中是可以正常同步的，在初始化RepLayOut的Cmds数组的时候，就会判断当前的属性类型是否是动态数组（UArrayProperty），并会给其cmd.type做上标记REPCMD\_DynamicArray。后面在同步的时候，就会通过这个标记来对其做特殊处理。比如服务器上数组长度发生变化，客户端在接收同步过来的数组时，会执行FRepLayout::ReceiveProperties\_DynamicArray\_r来处理动态数组。这个函数里面会矫正当前对象同步数组的大小。

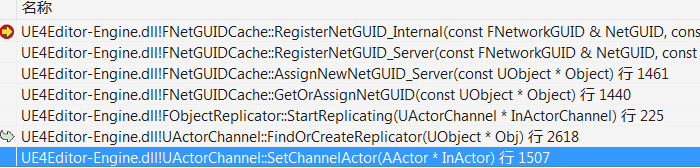
**7.UObject指针类型的属性同步**

上一节组件同步提到了FNetworkGUID，这引申出一个值得思考的细节。无论是属性同步，还是作为RPC参数。我们都可能产生疑问，我在传递一个UObject类型的指针时，这个UObject在客户端存在么？如果存在，我如何能通过服务器的一个指针找到客户端上相同UObject的指针？

这个处理就需要通过FNetworkGUID了。服务器在同步一个对象引用（指针）的时候，会给其分配专门的FNetworkGUID并通过网络进行发送。客户端上通过识别这个ID，就可以找到对应的UObject。

那么这个ID是什么时候分配的？如何发送的呢？

首先我们分析服务器，服务器在同步一个UObject对象时（包括属性同步，Actor同步，RPC参数同步三种情况），他都需要对这个对象进行序列化（UPackageMapClient::SerializeObject），而在序列化对象前，要检查GUID缓存表（TMap<FNetworkGUID, FNetGuidCacheObject>ObjectLookup;），如果GUID缓存表里面有，证明已经分配过，反之则需要分配一个GUID，并写到数据流里面。不过一般来说，GUID分配并不是在发送数据的时候才进行，而是在创建FObjectReplicator的时候（如图通过NetDriver的GuidCache分配）

图5-11 GUID的分配与注册

下面两段代码是服务器同步对象前检测或分配GUID的逻辑：

//UPackageMapClient::SerializeObjectPackageMapClient.cpp

//IsSaving表示序列化，即发送流程IsLoading表示反序列化，即接收流程

//由于知乎有字数限制，这里不粘贴完整代码

if (Ar.IsSaving())

{

//获取或分配GUID

FNetworkGUID NetGUID = GuidCache->GetOrAssignNetGUID(Object );

if (OutNetGUID)

{

\*OutNetGUID = NetGUID;

}

......

}

// PackageMapClient.cpp

FNetworkGUIDFNetGUIDCache::GetOrAssignNetGUID(constUObject \* Object )

{

//查看当前UObject是否支持网络复制

if( !Object || !SupportsObject( Object) )

{

return FNetworkGUID();

}

......

//服务器注册该对象的GUID

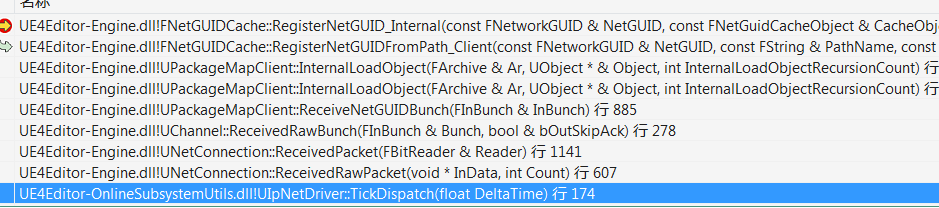
return AssignNewNetGUID\_Server( Object );

}

下面我们再分析客户端的接收流程，客户端在接收到服务器同步过来的一个Actor时他会通过UPackageMapClient::SerializeNewActor对该Actor进行反序列化。如果这个Actor是第一次同步过来的，他就需要对这个Actor进行Spawn，Spawn结束后就会调用函数FNetGUIDCache::RegisterNetGUID\_Client进行客户端该对象的GUID的注册。这样，服务器与客户端上“同”一个对象的GUID就相同了。下次，服务器再同步指向这个Actor的指针属性时就能正确的找到客户端对应的对象了。

不过等等，前面说的UObject，这里怎么就直接变成Actor了，如果是组件同步呢？他的GUID在客户端是怎么获取并注册的？

其实对于一个非Actor对象，客户端不需要在接收到完整的对象数据后再获取并注册GUID。他在收到一个函数GUID的Bunch串时就可以立刻执行GUID的注册，然后会通过函数FNetGUIDCache::GetObjectromNetGUID去当前的客户端里面寻找这个对象。找到之后，再去完善前面的注册信息。为什么要找而不是让服务器同步过来？因为有一些对象不需要同步，但是我们也知道他在客户端与服务器就是同一个UObject，比如地图里面的一座山。这种情况我们稍后讨论

图5-12 客户端收到消息立刻按照路径注册GUID

下面两段代码是客户端反序列化获取并注册GUID的逻辑：

// 情况一：客户端接收到服务器同步过来的一个新的Actor，需要执行Spawn spawn 成功后会执行RegisterNetGUID\_Client进行GUID的注册

// UActorChannel::ProcessBunch DataChannel.cpp

bool SpawnedNewActor = false;

if( Actor == NULL)

{

......

SpawnedNewActor = Connection->PackageMap->SerializeNewActor(Bunch,this,NewChannelActor);

......

}

// 情况二：客户端接收到一个含有GUID的消息立刻解析 解析成功后会执行RegisterNetGUIDFromPath\_Client进行GUID的注册

//DataChannel.cpp

void UChannel::ReceivedRawBunch(FInBunch&Bunch, bool&bOutSkipAck)

{

if( Bunch.bHasGUIDs )

{

Cast<UPackageMapClient>( Connection->PackageMap)->ReceiveNetGUIDBunch( Bunch );

......

}

}

// UPackageMapClient::ReceiveNetGUIDBunchPackageMapClient.cpp

int32 NumGUIDsRead = 0;

while(NumGUIDsRead <NumGUIDsInBunch )

{

UObject \* Obj = NULL;

InternalLoadObject(InBunch,Obj, 0 );

......

}

上面大部分讨论的都是标记Replicate的Actor或组件，但是并不是只有这样的对象才能分配GUID。对于直接从数据包加载出来的对象（前面说过如地图里面的山），我们可以直接认为服务器上的该地形对象与客户端上对应的地形对象就是一个对象。所以，我们看到还存在其他可以分配GUID的情况，官方文档上有介绍，我这里直接总结出来：

有四种情况下UObject对象的引用可以在网络上传递成功。

1. 标记replicate
2. 从数据包直接Load
3. 通过Construction scripts添加或者C++构造函数里面添加

使用UActorComponent::SetNetAddressable标记（这个只针对组件，其实蓝图里面创建的组件默认就会执行这个操作）

下面这段代码展示了该UObject是否支持网络复制的条件，正好符合我上面的总结：

//PackageMapClient.cpp

boolFNetGUIDCache::SupportsObject(constUObject \* Object )

{

if( !Object )

{

return true;

}

FNetworkGUID NetGUID = NetGUIDLookup.FindRef(Object);

//是否已经分配网络ID

if( NetGUID.IsValid() )

{

return true;

}

//是否是数据包加载或者默认构造的

if( Object->IsFullNameStableForNetworking())

{

return true;

}

//不重载的情况下还是会走到IsFullNameStableForNetworking里面

if( Object->IsSupportedForNetworking() )

{

return true;

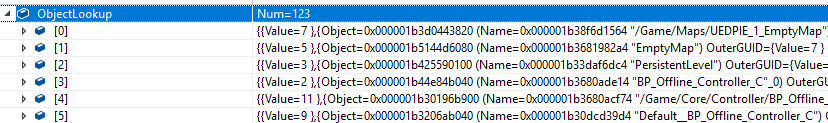
}

return false;

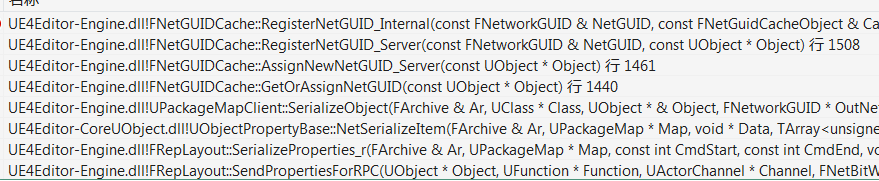
}

我这里以地图里面的静态模型为例简单进行分析。对于地图创建好的一个静态模型，服务器只要发送该对象GUID以及对象的名称（带序号）即可。当客户端接收消息的时候，首先缓存GUID相关信息，随后通过函数FNetGUIDCache::GetObjectromNetGUID从本地找到对应的Object。（如图5-13里ObjectLookup[24]对应的StaticMeshActor\_20，他就是一个非Replicate但是从数据包直接加载的对象）

下图5-13可以看出，分配GUID的对象不一定是游戏场景中存在的Actor，还可能是特定路径下某个资源对象，或者是一个蓝图类，或是一个CDO对象。进一步分析，一个在游戏里面实际存在的Actor想要同步的话，我们必须先将其资源文件，CDO对象先同步过去。然后再将实际的Actor同步，因为这样他才能正确的根据资源文件Spawn出来。而对于一个Actor的组件来说，他也需要等到他的Actor的资源文件，CDO对象先同步过去再进行同步。（由于网络包的异步性，这里并不是严格意义上的先后，而是指资源，CDO同步后，后面的Actor（组件）才能正常的反序列化成一个完整合法的对象）

图5-13 GUID缓存Map

最后再给出一个UObject作为RPC的参数发送前的GUID分配堆栈：

图5-14

关于动态组件的复制可能会有一些问题：我在文章[《Exploring in UE4》关于网络同步的理解与思考[概念理解]](https://zhuanlan.zhihu.com/p/34721113) 的最后有一些相关描述。

**8.UObject指针类型的属性更新**

一般来说，网络环境良好的情况下，网络包是按照我们发送的顺序反序列化的。但即使如此，我们还是会遇到各种与网络同步顺序相关的问题，大部分的问题都出在UObject类型的属性同步。

举个例子：服务器上有A和B两个replicated的Actor，A对象保存一个replicated的B类型属性，B对象保存一个A对象的replicated属性。A在同步的时候由于B没有同步过去，所以A的属性临时为空。当B同步到客户端的时候，会先执行Beginplay，但是这时候A对象里面的B属性仍然为空。

这样就会导致B::Beginplay里面找到属性A并尝试去处理A里面的B属性的相关逻辑失效。

那么为什么会这样？

首先，客户端每次收到数据包的时候会执行FObjectReplicator::ReceivedBunch去处理Bunch，这个函数有一个引用参数bOutHasUnmapped，函数会根据数据包的类型去处理Actor的Property或者RPC的Property，如果在反序列化的时候客户端上没有找到匹配的UObject，就会在引用参数里面标记bOutHasUnmapped并在UActorChannel::ProcessBunch里面将当前的actor的Replicator放到NetDriver的UnmappedReplicators数组里面。

拿Object类型的同步属性具体一点来说，FObjectReplicator::ReceivedBunch执行时会去尝试去反序列化该属性对应的UOBject（UPackageMapClient::SerializeObject），为了保持客户端与服务器对应属性的一致性服务器也会把GUID一同发送（Outer的GUID也可能在这个包里面），所以属性的GUID会在此刻被提前注册，虽然GUID可以注册成功，但是UObject对应的对象未必同步完成。如果反序列化失败就会把当前的GUID存放到TrackedUnmappedNetGuids里面。随后，就会判断TrackedUnmappedGuids的num是否大于0，是的话就会返回参数bOutHasUnmapped为true，另外还有一个引用参数bOutGuidsChanged表示其unmapped的GUID是否在同步过程中发生变化（比如该属性的Uobject对象发生改变）如果产生变化会调用UpdateGuidToReplicatorMap更新Netdiver的GuidToReplicatorMap，（关于GuidToReplicatorMap后面有进一步的描述）

其次，每个Actor在执行SerializeNewActor反序列化的时候都会将GUID放到GuidCache->ImportedNetGuids里面，表示当前帧新收到的Actor。（注册GUID的时候还会将GUID与对应的Object放到FNetGUIDCache的ObjectLookup里面。）每帧会在处理完收包以及所有对象的Tick之后通过TickFlush发送数据包，在发送之后会去处理unmapped的actor，这里会有一个比较复杂的筛选流程，

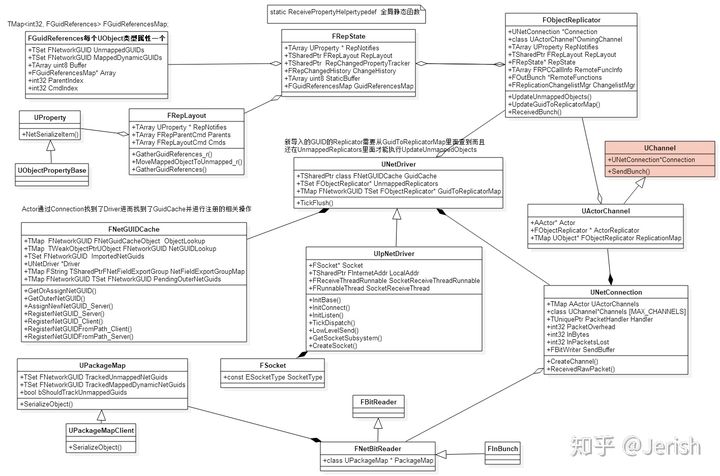
1.根据ImportedNetGuids确定当前帧所有新收到的actor  
2.去全局的GuidToReplicatorMap里面查找新收到的actor是否属于之前unmapped的GUID，是的话，假如待处理数组  
3.二次验证，根据UnmappedReplicators数据查找待处理数组是否处于其中，是的话执行Replicator->UpdateUnmappedObjects进行更新

筛选完成后通过函数FRepLayout::UpdateUnmappedObjects\_r去遍历所有属性的FGuidReferences，判断是否有UnmappedGUIDs，有的话调用PackageMap->GetObjectFromNetGUID重新尝试获取unmapped的GUID，如果反序列化仍然失败，那么就保存其UnmappedReplicators，不做任何处理。如果成功，就会覆盖原来的值。并且如果当前Actor的所有属性的GUID都已经成功匹配，就会把自己从NetDriver的UnmappedReplicators移除。

处理完后调用UpdateGuidToReplicatorMap，再次遍历一下所有的UObject类型的同步属性，如果发现有新的unmapped的属性（比如新的RPC函数有UObject参数，或者属性发生改变），就会把当前unmapped属性的GUID以及当前对象的Replicator放到NetDriver的GuidToReplicatorMap里面，GuidToReplicatorMap是一个value为TSet< FObjectReplicator\* >类型的数据结构。value为set的原因是当前unmapped属性可能在其他的ACtor里面也被引用，这样一个Unmapped的GUID可能会对应多个actor的Replicator。所以GuidToReplicatorMap是NetDriver控制的一个全局的对象，用来记录当前所有的unmapped对象的GUID以及他所在的actor。

这里的逻辑用前面的例子通俗一点来讲，A执行Beginplay的时候B没有同步过来，但是由于A有B的GUID，所以会先注册B的GUID，同时A的replicator会放到UnmappedReplicators并且将B的GUID与A的replicator组合成一个pair放到GuidToReplicatorMap里面。随后，B同步的时候会先通过收包执行Beginplay，Tick，最后执行发包的TickFlush。TickFlush的结尾，会通过ImportedNetGuids知道当前B是新来的对象，由于B在前面被放到了GuidToReplicatorMap里面，所以这里会找到A的replicator并执行更新，同时B也会去遍历看看有没有其他类型的Unmapped的对象，如果有会执行和前面A类型的操作。

所以，这个问题与整个引擎的逻辑执行顺序有关  
1.TickDispatch收包  
2.World对象Tick  
3.TickFlush收包  
对象的Beginplay执行在第一步，而unmapped的属性调整在第三步。



**补充：Bunch缓存**

有A和B两个Actor，A有一个replicated的属性B。

当A对象同步到客户端而B属性没有同步的时候，仍然会给B创建一个Channel的。而且这个数据包里面会有B的GUID，所以也会去给B注册GUID。

如果这个时候，客户端先收到了B的RPC或者其他属性的Bunch包，由于Actor还没有初始化（也就是GuidCache->IsGUIDLoaded为false）就需要临时把这个bunch缓存起来，放到ACtorChannel的queuebunches里面，同时把当前的ACtorChannel添加到当前Connection的TickChannel里面。（还有一种情况是他最先收到一个B的属性变化bunch，GUID甚至还没有注册）

每帧在发送数据包的时候（这时候已经确保收到了这一帧的所有网络包），去遍历所有的TickChannel的queuebunches，如果当前的ACtor已经被成功的反序列化就可以正常的处理这些queuebunches。

除了解决网络延迟造成的顺序不一致问题，我们也可以自定义的将某些包缓存存起来，通过函数ShouldQueueBunchesForActorGUID即可对某个特殊的Actor做bunch缓存，虚幻内置的回放系统就是这样处理的，在加载一个存档点的时候会有大量的数据包，为了避免卡顿，可以将大部分的包缓存起来逐帧处理。

**9.条件复制**

默认情况下所有的属性都是要同步的，除非我们设置条件复制。

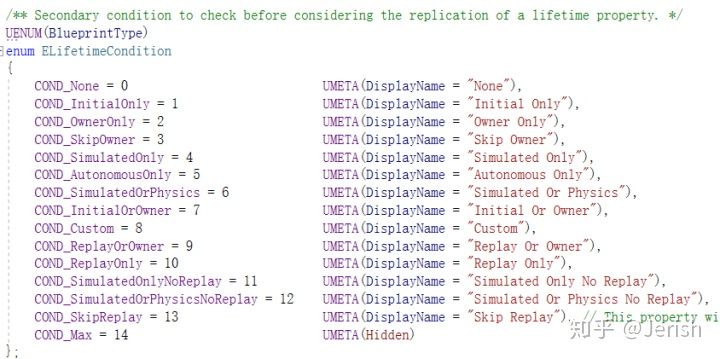
void AActor::GetLifetimeReplicatedProps( TArray< FLifetimeProperty > & OutLifetimeProps ) const

{

DOREPLIFETIME\_CONDITION( AActor, ReplicatedMovement, COND\_SimulatedOnly );

}

上面的代码表示我们只在模拟客户端进行ReplicatedMovement属性的同步。当然，具体的同步控制条件还有很多。



如果想更自由的控制条件复制，就需要使用COND\_Custom，并在当前Actor的PreReplication函数里面使用DOREPLIFETIME\_ACTIVE\_OVERRIDE，通过最后一个参数控制是否要进行同步。

DOREPLIFETIME\_ACTIVE\_OVERRIDE(AActor, ReplicatedMovement, bUseReplicatedMovement);

条件复制的基本原理是通过宏展开获取当前该属性的UProperty，然后封装放到参数OutLifetimeProps里面。

#define DOREPLIFETIME(c,v) \

{ \

static UProperty\* sp##v = GetReplicatedProperty(StaticClass(), c::StaticClass(),GET\_MEMBER\_NAME\_CHECKED(c,v)); \

for ( int32 i = 0; i < sp##v->ArrayDim; i++ ) \

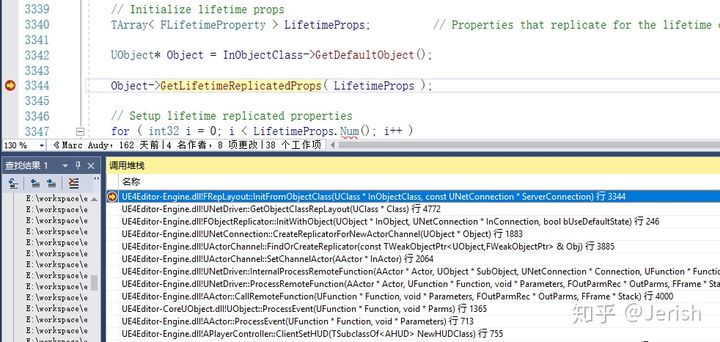
{ \

OutLifetimeProps.AddUnique( FLifetimeProperty( sp##v->RepIndex + i ) ); \

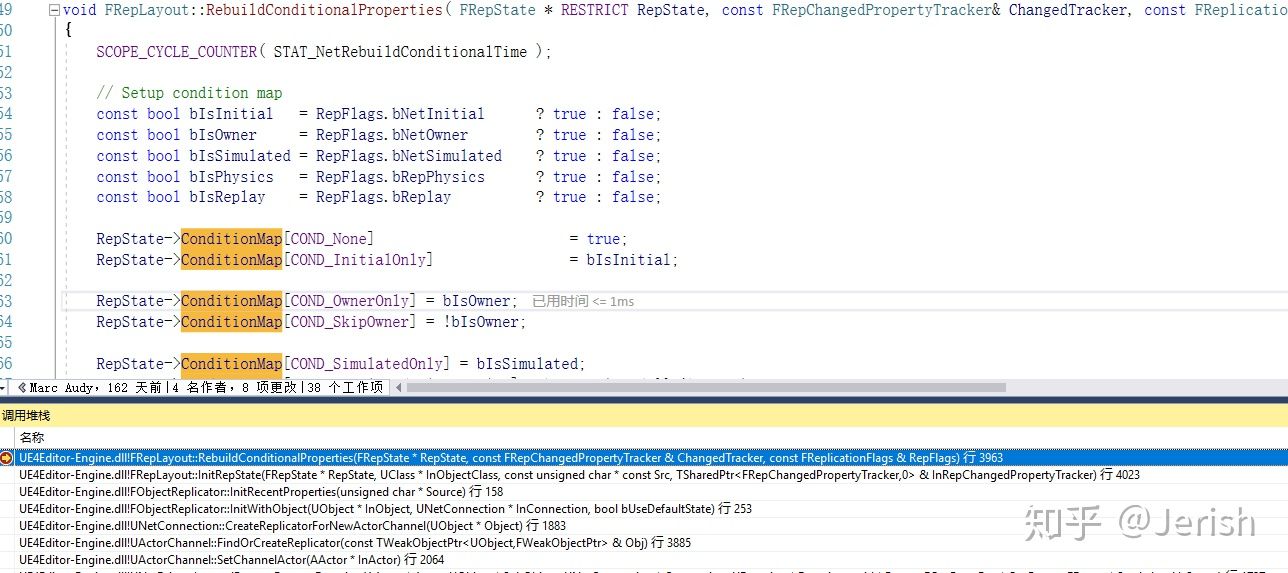
} \

}

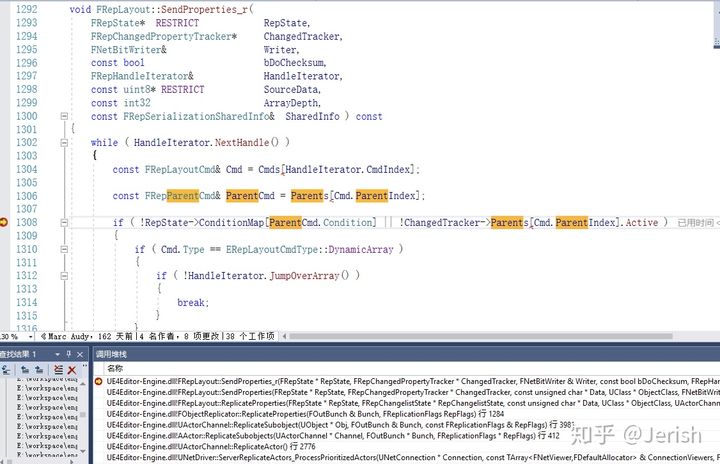
在每个ACtor的Replayout初始化的时候会去真正的调用GetLifetimeReplicatedProps并将LifetimeProps标记到对应的要同步的Parent数组里面（所以如果我们不在函数GetLifetimeReplicatedProps里面标记同步属性，这个属性即使标记了replicated也不会同步），同时LifetimeProps会把当前属性是否是条件复制（FRepParentCmd.Condition）放到Parent对应的标记里面。



另一边，在RepState初始化的时候会有一个ConditionMap来记录当前的条件复制情况。ConditionMap就是一个大小为enum ELifetimeCondition长度的bool数组，记录当前的复制条件。



现在，当我们正式进入序列化发送的时候，我们开始执行判断这个属性是否应该发送。判断方式就是找到当前属性的FRepParentCmd.Condition同时查看RepState的ConditionMap是否满足当前的条件。



从上面的堆栈可以看出，如果这时候不满足条件的话，当前的属性就不会写到序列化的bunch里面也就不会发送到当前的Connection里面。

**六． RPC执行细节**

RepLayOut参照表不止同步的对象有，函数也同样有，RPC的执行同样也是通过属性同步的这个框架。比如我们在代码里面写了一个Client的RPC函数ClientNotifyRespawned，那UHT会给我们生成一个.genenrate.cpp文件，里面会有这个函数的真正的定义如下：

void APlayerController::ClientNotifyRespawned(class APawn\* NewPawn, bool IsFirstSpawn)

{

PlayerController\_eventClientNotifyRespawned\_Parms Parms;

Parms.NewPawn=NewPawn;

Parms.IsFirstSpawn=IsFirstSpawn ? true : false;

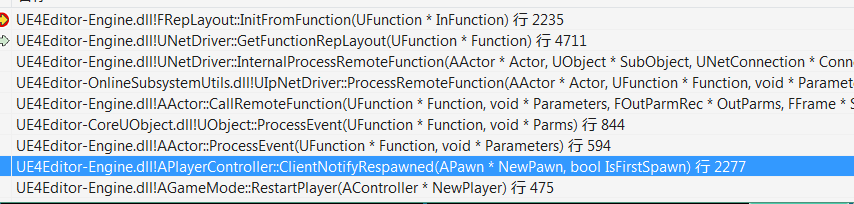
ProcessEvent(FindFunctionChecked(ENGINE\_ClientNotifyRespawned),&Parms);

}

而我们在代码里的函数之所以必须要加上\_Implementation，就是因为在调用端里面，实际执行的是.genenrate.cpp文件函数，而不是我们自己写的这个。

结合下面的RPC执行堆栈，我们可以看到在Uobject这个对象系统里，我们可以通过反射系统(通过名字在UClass里面查找，里面有一个存储TFunction的map)查找到函数对应的UFuntion结构，同时利用ProcessEvent函数来处理UFuntion。通过识别UFunction里面的标记，可以知道这个函数是不是一个RPC函数，是否需要发送给其他的端。 当我们开始调用CallRemoteFunction的时候，RPC相关的初始化就开始了。NetDiver会进行相关的初始化，并试着获取RPC函数的Replayout，那么问题是函数有属性么？正常来说，函数本身就是一个执行过程，函数名是一个起始的执行地址，他本身是没有内存空间，更不用说存储属性了。不过，在UE4的反射系统里面，函数可以被额外的定义为一个UFunction，从而保存自己相关的数据信息。RPC函数的参数就被保存在UFunction的基类Ustruct的属性链表PropertyLink里面，RepLayOut里面的属性信息就是从这里获取到的。 一旦函数的RepLayOut被创建，也同样会放到NetDiver的RepLayoutMap里面。随后立刻调用FRepLayout::SendPropertiesForRPC将RPC的参数序列化封装与RPC函数一同发送。

RPC中如果有指针参数，当RPC在客户端执行的时候，如果这个指针对应的actor还未在客户端生成，那么这个时候客户端拿到的RPC指针参数就是NULL。我们可以通过设置控制台变量net.DelayUnmappedRPCs 1允许客户端等到这个对象生成的时候再去执行，但是仅限于可靠的RPC。

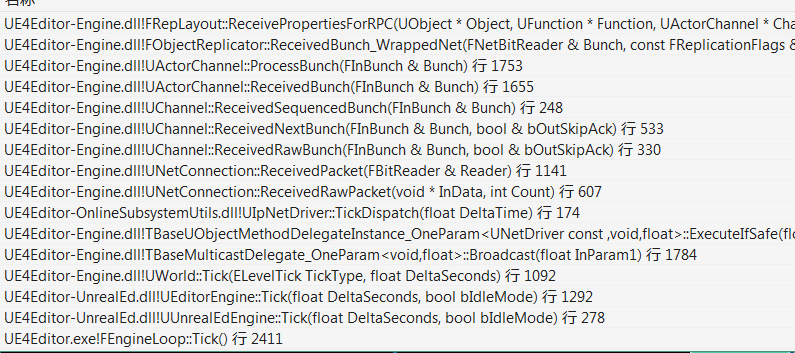
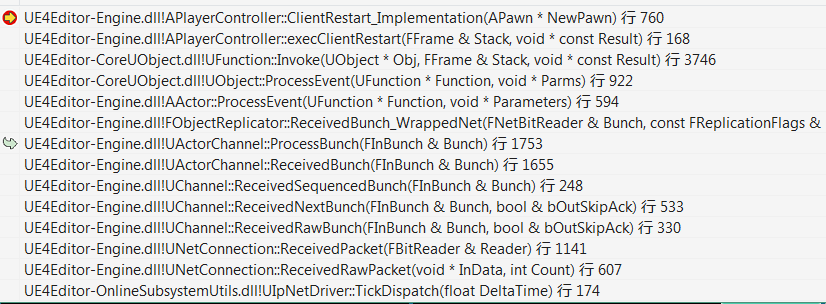
图6-1 RPC函数的RepLayOut初始化堆栈图

关于RPC的发送，有一个地方需要特别注意一下，就是UIpNetDriver::ProcessRemoteFunction函数。这个函数处理了RPC的多播事件，如果一个多播标记为Reliable，那么他默认会给所有的客户端执行该多播事件，如果其标记的是unreliable，他就会检测执行该RPC的Actor与各个客户端的网络相关性，相关才会执行。简单总结来说，就是一般情况下多播RPC并不一定在所有的客户端都执行，他应该只在同步了触发这个RPC的Actor的端上执行。

//UIpNetDriver::ProcessRemoteFunction

//这里很详细的阐述UE这么做的原因

简单概括了RPC的发送，这里再说一下RPC的接收。当客户端收到上面的RPC发来的数据后，他需要一步一步的解析。首先，他会执行ReceivePropertiesForRPC来接收解析RPC函数传来的参数并做一些判断确定是否符合执行条件，如果符合就会通过ProcessEvent去处理传递过来的属性信息，找到对应的函数地址（或者说函数指针）等，最后调用该RPC函数。 这里的ReplayOut里面的Parents负责记录当前Function的属性信息以及属性位置，在网络同步的过程中，客户端与服务器保存一个相同的ReplayOut，客户端才能在反序列化的时候通过OffSet位置信息正确的解析出服务器传来的RPC函数的N个参数。

图6-2 接收RPC函数的传递的参数堆栈图图6-3 客户端执行RPC函数堆栈图

最后客户端是怎样调用到带\_Implementation的函数呢？这里又需要用到反射的机制。我们看到UHT其实会给函数生成一个.genenrate.h文件，这个文件就有下面这样的宏代码，把宏展开的话其实就是一个标准的C++文件，我们通过函数指针最后找到的就是这个宏里面标记的函数，进而执行我们自己定义的\_Implementation函数。

virtual void ClientNotifyRespawned\_Implementation(class APawn\* NewPawn, bool IsFirstSpawn);\

DECLARE\_FUNCTION(execClientNotifyRespawned) \

{ \

P\_GET\_OBJECT(APawn,NewPawn); \

P\_GET\_UBOOL(IsFirstSpawn); \

P\_FINISH; \

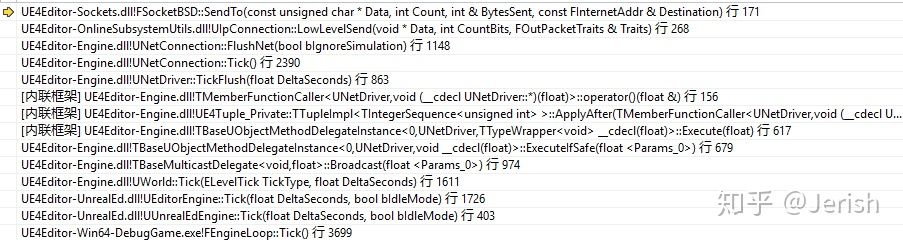
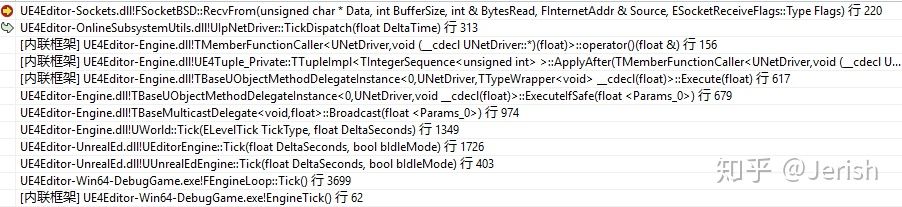
this->ClientNotifyRespawned\_Implementation(NewPawn,IsFirstSpawn); \

} \

RPC的数据包相比属性同步不同，是调用的时候立刻产生的Bunch并放到Sendbuffer里面的，按照UE4一帧的执行顺序（收包-Tick-发包），一般RPC的数据要比属性同步要早放到buffer里面，所以经常出现RPC与属性同步顺序不对导致的同步问题。

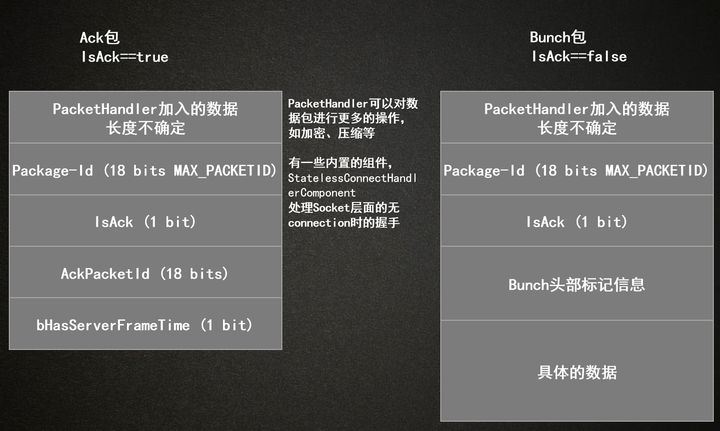
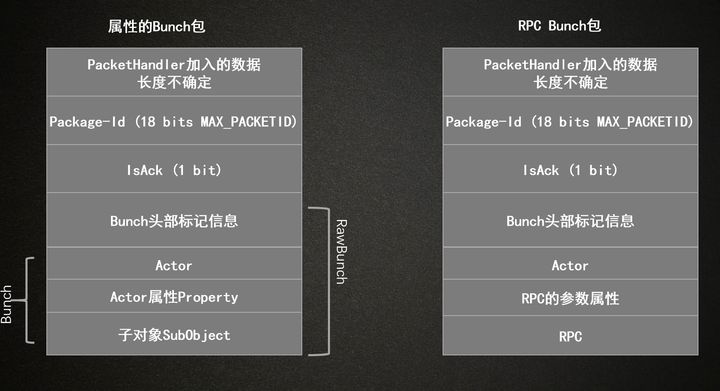
**七． 可靠数据传输**

UE4默认收发包都在主线程处理，收包可以通过控制CVarNetIpNetDriverUseReceiveThread来开启线程单独处理。

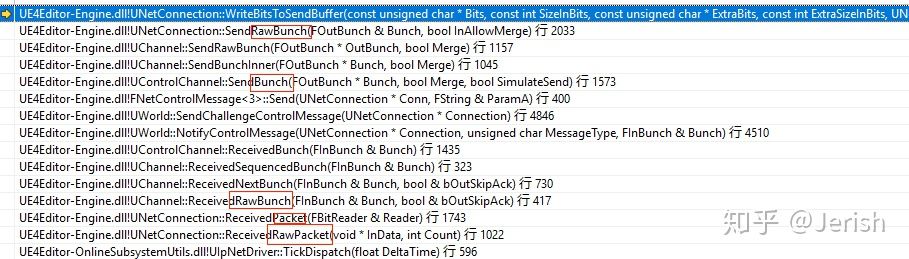
发包堆栈收包堆栈

**1.数据包格式**

这里再次拿出来一般网络数据包的格式，可以看到虚幻里面的网络包是精确到bit的，这些信息都可以通过FBitWriter与FBitReader去读与写。

网络包分为Ack与Bunch两种对于ActorChannel，Bunch分为属性Bunch与RPC Bunch

平时看函数堆栈的时候我们可能看到Bunch、RawBunch、Packet、RawPacket等。所谓的Bunch就是上面图所展示的（ActorChannel发送数据的Bunch分为属性Bunch与RPC Bunch），Bunch如果太大就会被拆分成很多个小的Bunch，一旦拆分成小的bunch那么这个bunch就不是一个完整的bunch（就可以叫做一个Rawbunch，具体逻辑在UChannel::SendBunchInner里面），这些bunch可以都被塞到一个Sendbuffer里面，如果这样直接发出去，就是一个Packet。每一个Sendbuffer发出前还可能会被PacketHandler处理，处理之后就是RawPacket。按照这样的理解，你就能看懂下面的堆栈了。

客户端与服务器通过ControlChannel建立连接的某次通信堆栈

另外，由于平时我们的发包都是按照最小单位Byte来发送的，UE4里面又精确到bit。所以会在Sendbuffer最后面添加1bit的结束标志位，另一端在收到包的时候就可以先找到最后一个为1的bit，把后面的0删除，前面剩下的就是原始的网络包。

**2.PacketHandler**

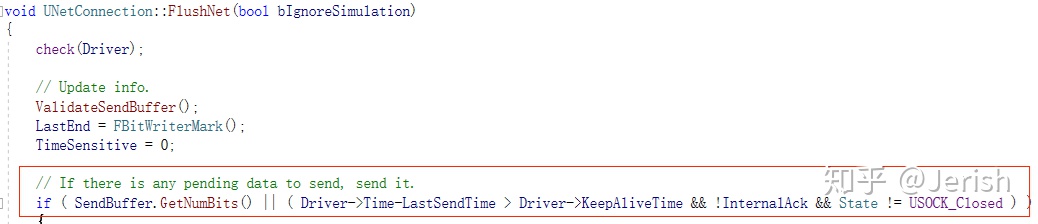
PacketHandler是用来对原始数据包（Packet）进行处理的一个“工具”，里面可以自由的添加组件来对原始数据包进行多层处理，目前引擎内置的组件有握手组件StatelessConnectHandlerComponent、各种加密组件FEncryptionComponent、可靠数据传输组件ReliabilityHandlerComponent等。由于组件的不确定性，所以网络的消息包头也是不确定的。比如加密组件可能会对一个Packet进行加密，然后在前面添加一个2bit的头部以及1bit的结束标志位，也因此各个组件应该固定的顺序处理packet。默认情况下回一直存在一个StatelessConnectHandlerComponent组件。

由于PacketHandle组件可能对原有的packet进行加密从而导致位发生变化，所以PacketHandle组件本身也会对处理过的数据后添加一个bit的结束标志位。



**3.Bunch的发送时机**

每次只要执行sendrawbunch（可能在netdrivertick里worldtick里面的代理tick，也可能在worldtick里tickgroup里面）就会设置TimeSensitive为true，就会触发flushnet，所以说只要每帧有数据就会发送。只要里面有sendbuffer或者到时间了就会触发lowlevelsend，调用socket的发送



**4.可靠数据传输的实现**

可靠数据传输的基本原理就是接收方对每一个包都要做Ack回应，如果接收方没收到Ack，那么就要进行重传。

UE4底层默认是主动重传，只要没有按顺序收到bunch就会重传。每个包有一个OutPacketId（记录在Connection里面），一个packet可能包括N个Bunch，每个bunch也会记录当前所在的OutPacketId。

简单来说，发送端会记录一个已经传送成功的序号（已经收到的Ack.OutAckPacketId）假如发送端发了10个包（1-10），接收端收到了1那么会回复一个ack，里面是OutAckPacketId 1。然后发生丢包，接收端收到了序号5，那么就会回复一个ack5，这时候发送端会更新当前的OutAckPacketId并重传序号2-4所有的packet（保存在connection的缓存里面）。所以，可以保证所有的包到上层都是严格有序的。

**if**( AckPacketId**>**OutAckPacketId )

{

**for** (int32 NakPacketId **=** OutAckPacketId **+** 1; NakPacketId**<**AckPacketId; NakPacketId**++**, OutPacketsLost**++**, OutTotalPacketsLost**++**, Driver**->**OutTotalPacketsLost**++**)

{

UE\_LOG(LogNetTraffic, Verbose, TEXT(" Received virtual nak %i (%.1f)"), NakPacketId, (Reader.GetPosBits()**-**StartPos)**/**8.f );

ReceivedNak( NakPacketId );

}

OutAckPacketId **=** AckPacketId;

}

除了Bunch里面的OutPacketId外，每个channel里面的还有一套ChSequenceID，记录了当前通道内可靠数据包的序号，每次发送加1。每个Connection里面会有N个Channel，每个Channel发出去的可靠数据包的数量会以Connection->OutReliable数组的形式存储，而真正发出去与接收到的数据包会缓存在OutRec链表与InRec链表链表里面，每次发送一个数据包就会添加到OutRec里面并设置其Ack状态为0，收到一个Ack的时候就会遍历当前Channel的OutRec链表，将对应Ack设为1，调用Channel::ReceivedAcks()并清空OutRec中被确认过的前面的所有缓存。OutRec并没有限制大小，所以理论上这里会出现内存溢出的情况，不过在逻辑上层还有一些自己的处理机制，比如Channel可以设置阈值，超过阈值就退化成停等协议，具体内容请参考UChannel::SendBunchInner。

每个通过有一个ChIndex，connection在接收Bunch的时候可以通过这个Index找到对应的Channel再下发消息。

**5.属性的可靠传输**

首先要确认一点，属性同步本身并不是可靠的，也就是他的属性bunch所在的packet如果丢失并不会将这个packet重新发送。只有Actor在第一次同步的时候才会设置合格属性bunch为Reliable

// Send initial stuff.

//UActorChannel::ReplicateActor

if( OpenPacketId.First != INDEX\_NONE && !Connection->bResendAllDataSinceOpen )

{ //第一次收到spawn的ack会把后面不可靠的属性也重新同步一遍

if( !SpawnAcked && OpenAcked )

{

// After receiving ack to the spawn, force refresh of all subsequent unreliable packets, which could

// have been lost due to ordering problems. Note: We could avoid this by doing it in FActorChannel::ReceivedAck,

// and avoid dirtying properties whose acks were received \*after\* the spawn-ack (tricky ordering issues though).

SpawnAck

ed = 1;

for (auto RepComp = ReplicationMap.CreateIterator(); RepComp; ++RepComp)

{

RepComp.Value()->ForceRefreshUnreliableProperties();

}

}

}

else

{ //第一次同步是可靠的

Bunch.bClose = Actor->bNetTemporary;

Bunch.bReliable = true; // Net temporary sends need to be reliable as well to force them to retry

}

那么属性是怎样做到可靠的呢？我发现即使接收方即使接收到的Packet里面的bunch不是reliable的，在通道不关闭、不是拆分的Bunch等情况下还是会回复一个Ack的，所以发送端可以接收到一个Ack从而知道当前的属性是否被另一端接收到。

当发生丢包或者乱序的时候，发送方会接收到乱序的ACK进而RepState会记录当前Nak的数量，并对当前的同步发送历史信息进行标记

void FRepLayout::ReceivedNak( FRepState \* RepState, int32 NakPacketId ) const

{

if ( RepState == NULL )

{

return; // I'm not 100% certain why this happens, the only think I can think of is this is a bNetTemporary?

}

for ( int32 i = RepState->HistoryStart; i < RepState->HistoryEnd; i++ )

{

const int32 HistoryIndex = i % FRepState::MAX\_CHANGE\_HISTORY;

FRepChangedHistory & HistoryItem = RepState->ChangeHistory[ HistoryIndex ];

if ( !HistoryItem.Resend && HistoryItem.OutPacketIdRange.InRange( NakPacketId ) )

{

check( HistoryItem.Changed.Num() > 0 );

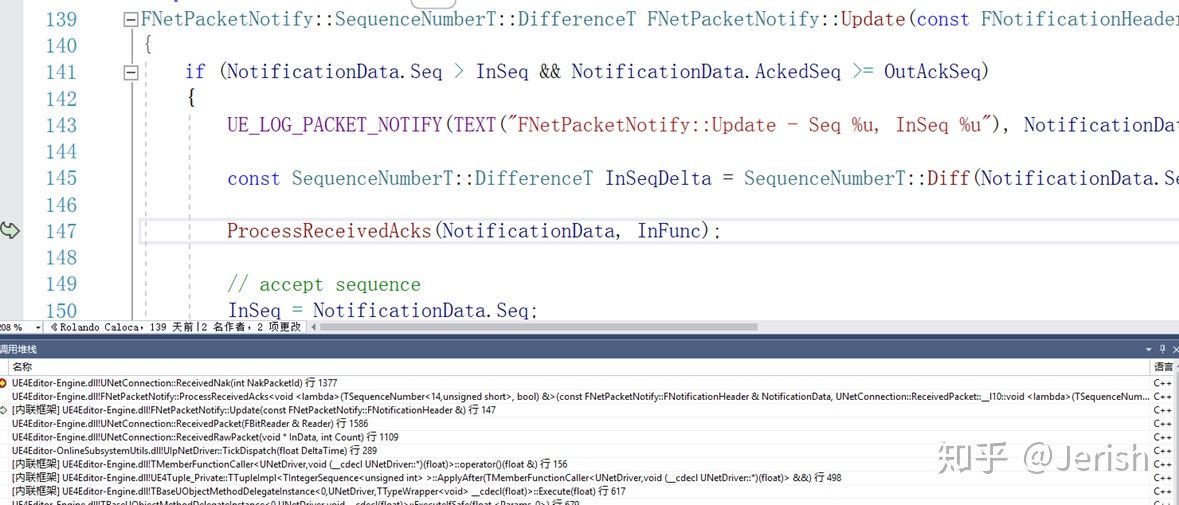
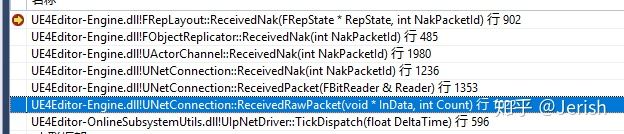
HistoryItem.Resend = true;

RepState->NumNaks++;

}

}

}



当下一帧要进行属性同步的时候，就会把之前的历史记录合并到最新的历史记录里面，然后一起发出去，也就是说丢失属性和新的属性会一起发送，这样达到了不用重发丢失的bunch还能保证属性可靠的效果了。这一块的逻辑主要在FRepLayout::ReplicateProperties里面，关于属性变化的历史记录可以参考上面第五章第4小节。

**八． ReplicationGraph**

ReplicationGraph是Epci官方针对堡垒之夜网络同步优化而加入的新的插件系统，可以大大减少Actor的同步与遍历，比较适合对大世界场景进行网络同步优化。这一块已经有[文章](https://zhuanlan.zhihu.com/p/56922476)写的比较清晰了，所以我只是简单的列举其优化点与基本原理。

通常服务器在同步Actor到各个连接的时候，会遍历场景中所有标记Replicated的Actor，但是实际上与玩家距离比较远的根本就不需要遍历，更不用说同步，所以ReplicationGraph加入了GridSpatialization2D节点系统，把N\*N的格子，并把Actor放到当前所有与他有关的格子里，这样一个玩家靠近他的时候就从当前子集所在的格子里面找一下有没有那个Actor就可以了（只遍历所有在这个格子里的Actor即可）



当然，作为一个系统不仅仅是提供这样一种功能和优化，其里面还内置了很多节点用于不同的同步需求（比如可以对不同的Connection进行某一个Actor的特定属性进行共享序列化），你也可以自定义一个节点专门处理某些需要特殊处理的Actor，严格控制它的同步时机。这一块可以参考官方的Shootergame项目。

注：ReplicationGraph是一个纯C++插件系统，使用的话需要修改配置文件DefaultEngine.ini里面的内容。

# UE4网络同步思考

**经典同步方案**

## 1. 什么是网络同步？

多人游戏里面需要把某个玩家操作的结果通知给其玩家，这个通知的过程就是同步，再放到局域网或者广域网中进行，就是网络同步了。

## 2. 点对点还是中介转发？

点对点（P2P）就是N台机器相互直连，每台机器需与剩下N-1个机器建立连接，复杂度为O(N^2)，而且增删机器也非常麻烦，对于大型网游来说这个方案基本不考虑。

中介转发就是我们常说的客户端/服务器模式（C/S），添加一个服务器作为中介节点，每台客户端只与服务器建立连接，客户端与客户端彼此独立，服务器负责转发消息。这种架构的复杂度仅为O(N)，维护起来也方便，所以网游几乎清一色的采用这套架构。

## 3. 帧同步还是状态同步？

帧同步是服务器将某个玩家的输入直接转发给其他玩家，自己不做处理。理论上所有客户端都以相同的初始状态开始，只要收到的输入相同，那么每时每刻的状态都会是相同的。

状态同步是服务器只同步影响游戏功能的某些重要状态变量，并且这些重要变量是在服务器运算出来的或者至少校验过的，客户端拿到这些状态变量后自行做本地的表现。

一般来说帧同步在实时性、节省流量方面比较好，状态同步则在安全性角度来说更胜一筹。具体选用哪种方案由具体游戏类型来决定，UE是在射击游戏基础上发展而来的，它默认的网络同步方案是状态同步，把决策权放在服务器上做，可以有效减少外挂，对于中途加入/断线重连也能天然支持。

## 4. TCP还是UDP？

TCP连接的优点是可靠稳定，但速度慢这个缺点导致它并不适合网游。但UE也并未照搬UDP连接方案，而是在这个基础上融合了TCP的优点，例如加入了乱序处理，以及对reliable的包丢失重传。可谓是各取所优，既保证了连接速度，也保证了可靠性。

## 5. UE如何同步重要状态变量？

保守的方法是把这些重要状态变量每时每刻都对所有客户端进行广播，但这样会造成流量的问题，也会带来客户端本地的计算复杂性。对于UE来说，需要同步的重要状态变量都存在于Actor这个容器里。UE网络同步的设计方案有以下两个特点：

（1）尽可能节省带宽：

相关性计算：对于像吃鸡这种大世界玩法，不需要每帧都同步一百个玩家的状态，对于A玩家而言，只需要同步他可以看到的或者会受到影响的那部分玩家就可以了，这就是相关性的计算。

优先级计算：判断相关后还会进行优先级计算，优先同步高优先级的状态变量，如带宽饱合则当帧不会同步低优先级的状态变量。

成员变量计算：需要同步的Actor里面会包含很多成员变量，但不是每个成员变量都需要同步，只有开发者打上网络同步标记的变量才会被同步，而且这个变量只有在服务器端发生变化了才会同步给相关的客户端，甚至还有额外标记控制同步，比如bNetInit只在刚建立同步通道时同步。

合包：每个包里都会有额外的包头信息，如果包拆得越细那么占用带宽就会越多，UE里面将多个Bunch（最小的同步单位）合入到一个Packet里面进行发送，再在收端分拆回Bunch进行逐个处理。

（2）尽可能可靠：

丢失重传：reliable的包在发端会保存一个备份，只有收到收端返回的Ack包确认后才会清掉，若收到的Ack包跳序则会触发重传。

Packet乱序整理：因网络链路复杂性，到达包顺序可能与发端不一致，这时会进行整理，对于分包导致的不完整的部分（Partial）包也会等待重组。

包头校验：包头存有内容块的大小，如果不符合，则会丢弃这个包，防止坏包带来的后续数据处理异常。

## 6. 成员变量同步还是远程函数（RPC）？

成员变量同步的持久性较好，RPC则是瞬发的，玩家断线重连后采用变量同步的方式仍是正确的，但RPC同步的东西却不会再重放。

成员变量同步仅发生在服务器到客户端，RPC则是双向的，如果想客户端同步给服务器什么东西，那没得选只能RPC。

RPC的实时性比成员变量好，因为RPC调用是瞬时发到远端执行的，而成员变量同步还要等packet满了之后才发车，不过这个因素其实不那么重要，两者相差很小。

## 7. 如何正确又优雅地使用UE网络同步功能？

提三点建议：

其一：一开始做功能先别管网络，优先单机，单机表现不正常剩下的都是白谈。

其二：大概看下UE官网对于网络同步的使用，知道有哪些关键字（不多很简单），再结合自身项目代码或者ue4的sample工程（如shooter game），看下别人如何使用的，理解后学以致用，写好后在编辑器里面选2人勾DedicatedServer跑下是否正常。值得一说的是，UE4新加了component的同步，要善用这个模块化的利器，不必像UE3一样把同步变量都堆在Actor上。

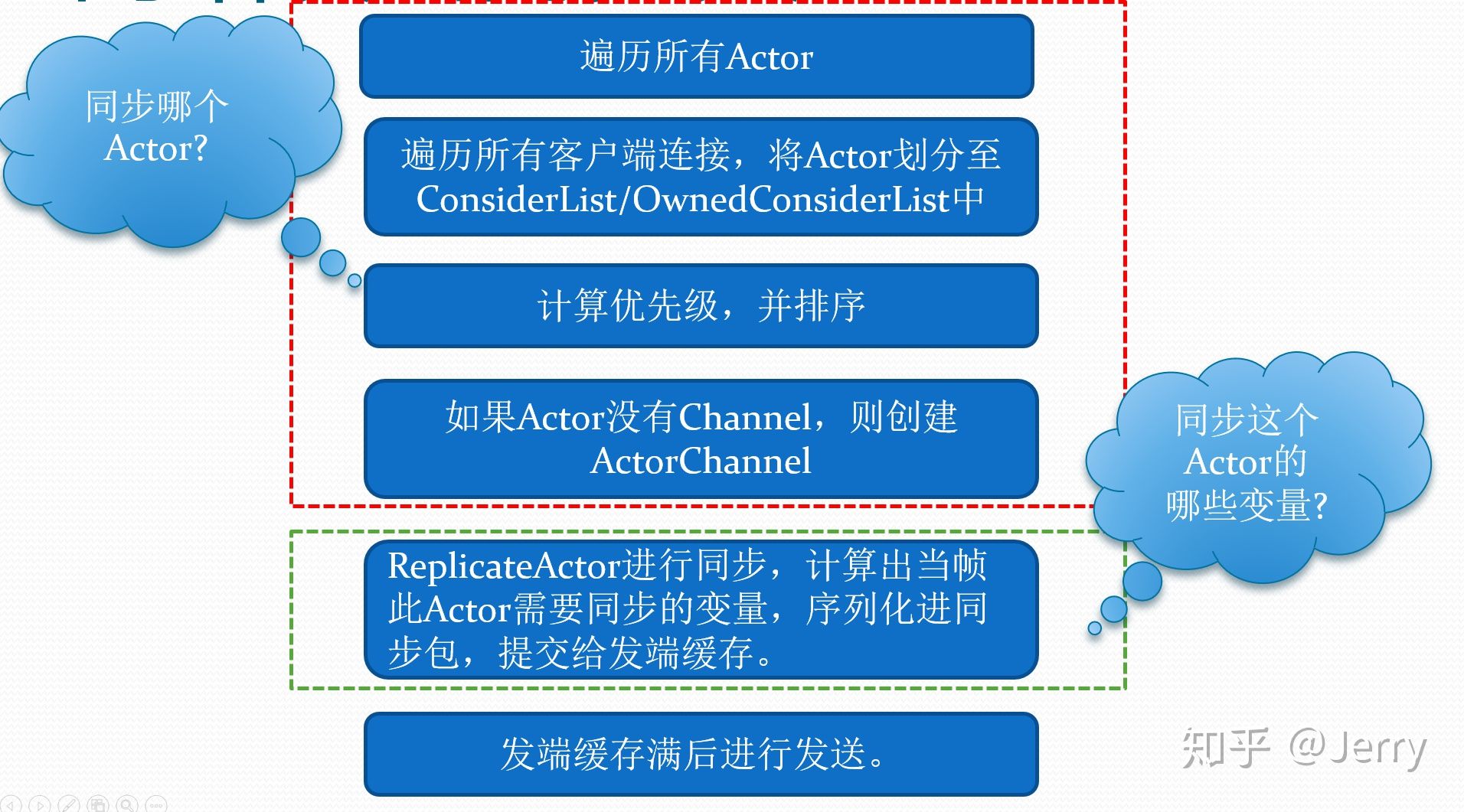
其三：知其然更要知其所以然，使用起来方便但背后设计却非常复杂，要静下心下跟跟代码，看看前面几位大神的文章，对网络实现细节深入理解。

UE4.20版本针对大世界推出了一套新的网络同步方案RepicationGraph，解决50000+对象同步问题。

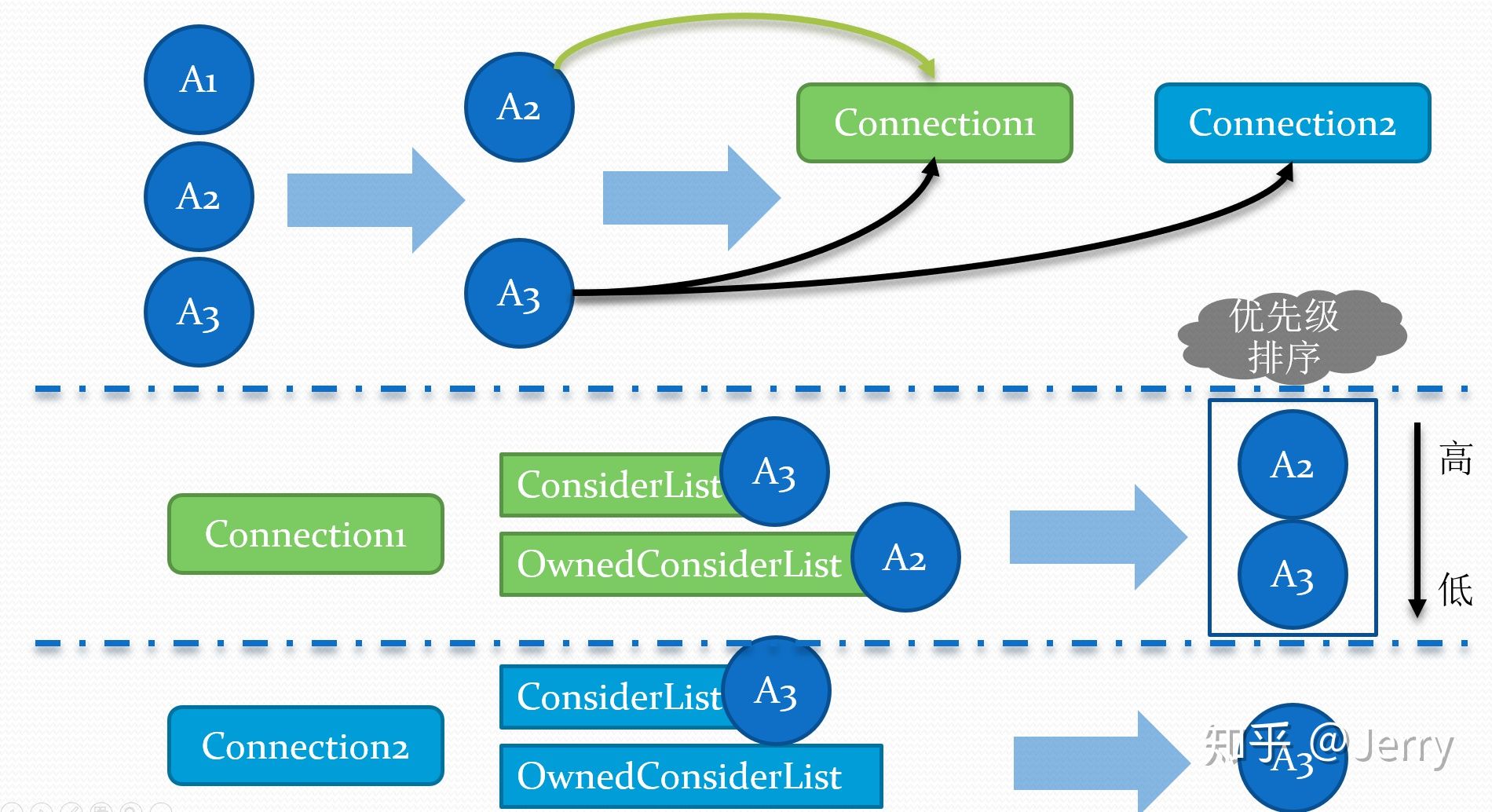
**大世界同步方案ReplicationGraph**

介绍了下UE经典网络同步方案，适用于绝大多数射击游戏同步场景，但眼下吃鸡玩法正兴，Fortnite大世界有多达50000+个同步对象，按照传统的同步方案会给服务器的CPU带来巨大的压力。

专用服务器（Dedicated Server，简称DS）CPU的消耗位于网络同步的发送端，因此我们先回顾下经典网络同步方案中的发端算法。



举个图例，DS每帧会遍历游戏世界里的所有对象，这里假定是A1,A2,A3，排除掉不需要进行同步或者不相关的A1，再根据A2和A3的同步属性划规到不同的连接通道Connection里面，这里A2与所有客户端连接都相关，而A3仅与Connection2相关。因同步带宽有限，只能保证高优先级的Actor最先同步，所以需要进行排序。排序后就是每个Actor内部的同步属性比较，以及将变化属性的地址和变化量写入包中并发送。



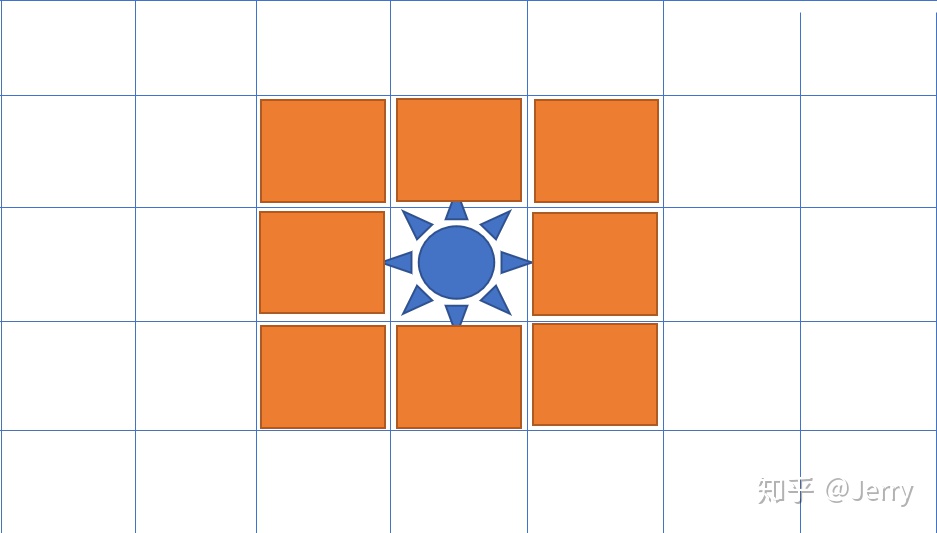
注意经典发端算法存在的几个问题：

(1)每帧遍历所有Actor，还要逐个计算每一个Actor与每一个客户端连接的相关性，即使这些相关性是确定的（如同队玩家）。

(2)一些场景中固定不动的同步对象，比如拾取物Pickup，房屋门窗（因有破坏性所以需同步）都会在每帧计算相关性，而客户端玩家在有限时间内不太可能有较大范围的空间移动。

以上，如能利用起大世界的空间相关性，就可以进一步优化网络同步的发端算法。

一种想法是将大世界网格化，每一个需要同步的Actor都会落在其中一个格子里，那么只要规划好每个格子的大小，就只需要同步玩家周围的九个格子里面的Actor即可，如下图所示橙色区域所示，还要包含玩家蓝色太阳自身所在的格子，形成九宫格：



这样对于静态物体，比如拾取物pickup，就不再需要每帧计算相关性了，只要有玩家当前所在位置，周围九宫格内（含自身）的Actor都是需要同步的。

对于动态物体，比如其他玩家，他们的位置每帧都可能在变化，仍可空间化到具体的栅格中来，只需要每帧更新玩家所在的栅格即可。

所以对一个客户端玩家来言，每帧需要做两件同步相关的事情：

(1)根据位置拉取它周围九宫格内的同步对象

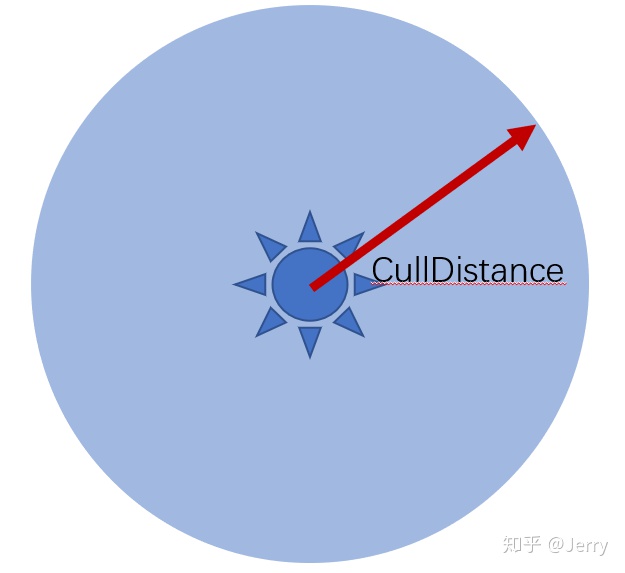
(2)自身位置变换时需更新所在栅格，要保证其他客户端连接能及时地同步自己。

上述算法已经考虑到了空间相关性，但每个格子划分多大也是要讲究的，划分太大则同步对象太多，CPU负担降不下来，划分太小又可能因为同步不及时带来体验上的问题（eg：被一个看不见的敌人杀掉）。

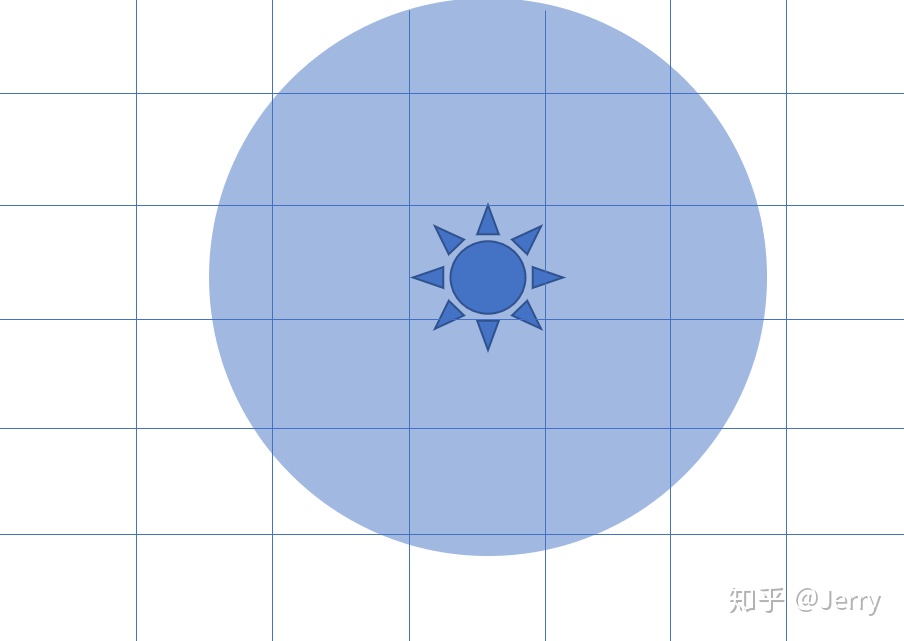
对于Pickup来说，因为大多集中在室内，所以同步距离可以少一些，但对于玩家来说，同步距离则要大到视距，这样格子划分的大小很难统一。保守做法就是统一按最大视距来划分格子，但这样优化量就很少。另一种做法是分门别类，根据类型划规不同大小的栅格，然后分开处理不同类型对象的同步，缺点就是存在多套栅格系统，是一种以空间换时间的做法。

UE4的ReplicatonGraph解决了上面的问题，相较于上述将一个对象放置于一个栅格中，转而设置这个对象的多个影响栅格，具体如下图示：

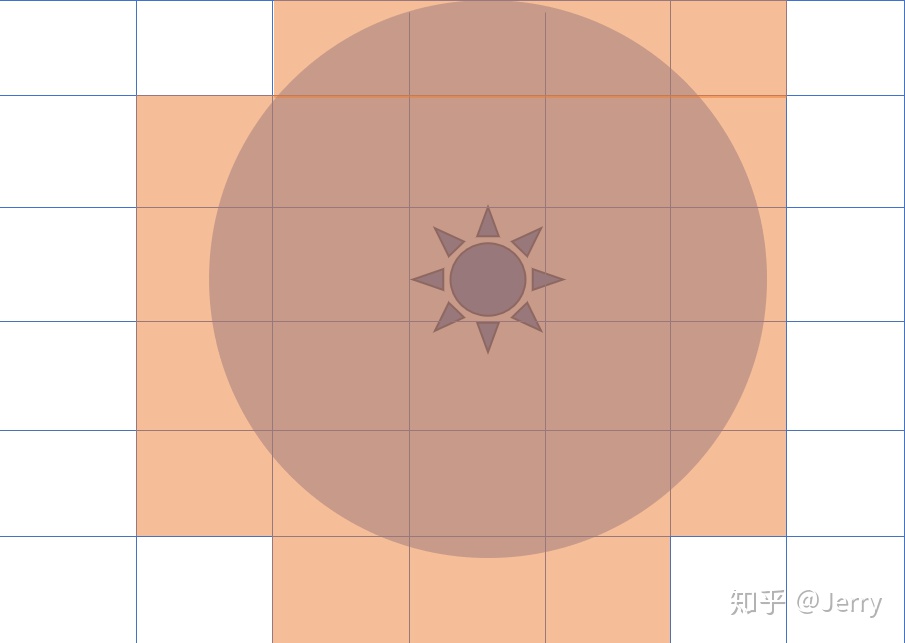
每个对象设置了一个同步相关的CullDistance，



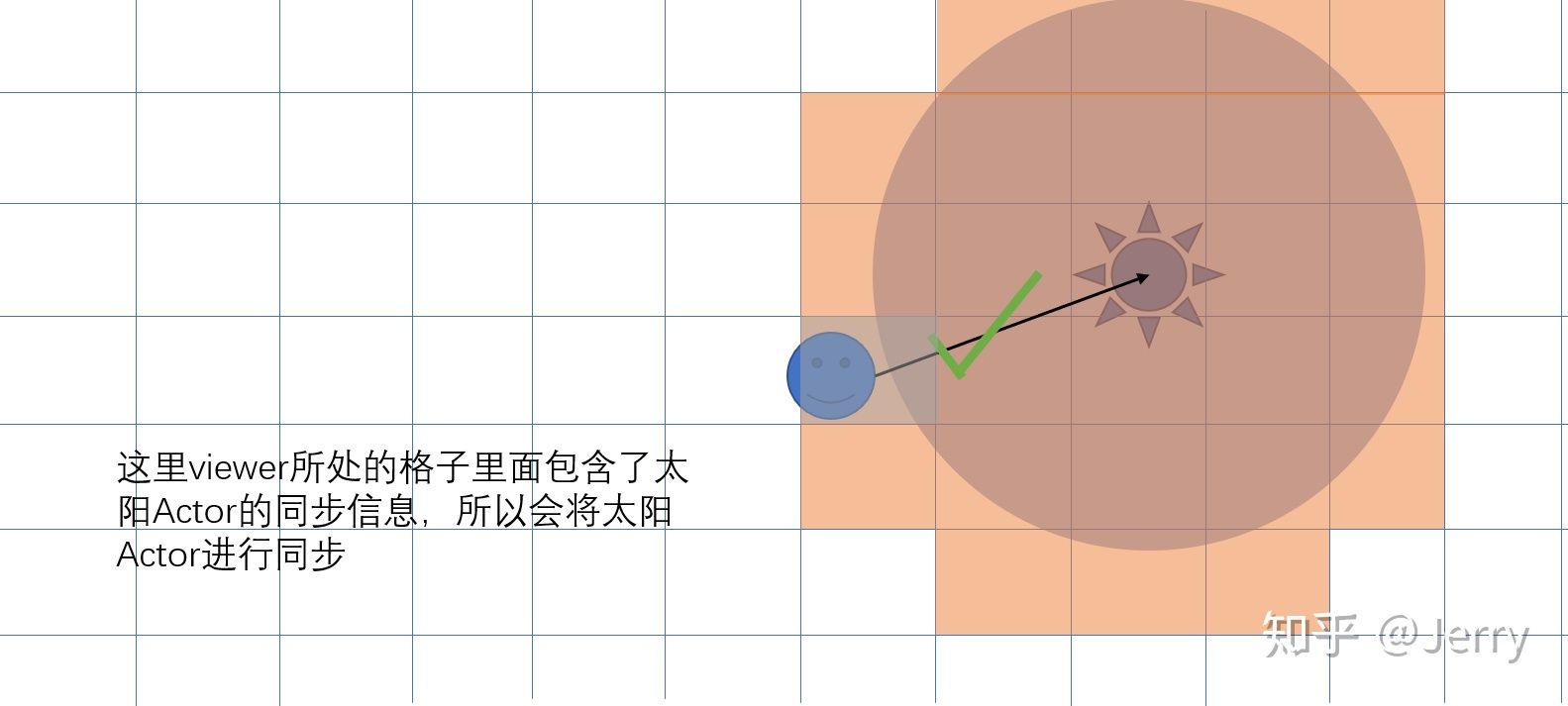
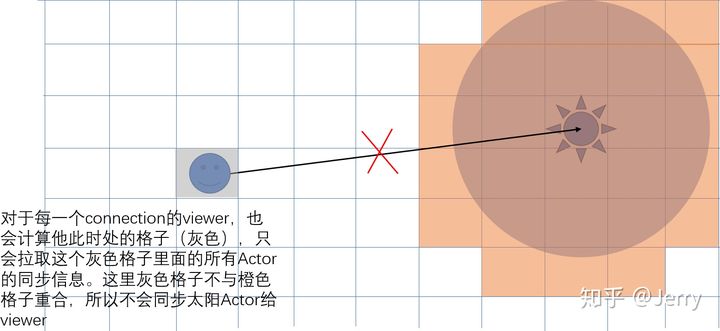
将之栅格化：



橙色区域为其影响的栅格：



只要客户端玩家走进了橙色栅格中的任意一个，都会收到这个太阳对象的同步信息。以下示例：



这个方案的优点是可以根据实际需要设置不同的CullDistance，同时保证了栅格自身大小可以固定不变。而且一般地，相同类型对象的CullDistance是相同的，即只要把CullDistance这个变量配置到类Class身上即可。

这样Class配置会很多吗？当父类Class与子类Class的关键同步属性相同时，只会考虑父类Class，所以不必担心。

同样分析下静态/动态物体的空间化同步，对于静态物体，位置总是固定的，因此不需要实时计算影响栅格，对于动态物体，因为空间相关性，短时间内栅格变化的概率也比较低，即使有变化，也只是少数几个，不必要每帧都新算一遍影响栅格，大家可以研究下UE4是如何增量计算栅格变化的。

以上是需要空间化的Class，还有一些诸如总是同步的对象，仅与某个客户端连接相关（或小队客户端连接相关）的对象，UE4都是分开处理的，具体大家可以参考官方示例ShooterGame里面的ReplicationGraph配置。

总结一下，采用ReplicationGraph充分利用了大世界Actor虽然多但却具有空间相关性的特性，并且考虑不同类型Actor的特性（总是相关的，与特定连接相关的，以及空间化相关的），有效减少了不必要的相关性计算，从而有效节省服务器CPU的负载。

# 关于网络同步的理解与思考[概念理解]

**一. 关于Actor与其所属连接**

[UE4官网](https://link.zhihu.com/?target=https%3A//docs.unrealengine.com/en-US/Gameplay/Networking/Actors)关于网络链接这一块其实已经将的比较详细了，不过有一些内容没有经验的读者看起来可能还是比较吃力。

按照官网的顺序，我一点点给出我的分析与理解。首先，大家要简单了解一些客户端的连接过程。

主要步骤如下：  
1.客户端发送连接请求  
2.服务器将在本地调用 AGameMode::PreLogin。这样可以使 GameMode 有机会拒绝连接。  
3.如果服务器接受连接，则发送当前地图  
4.服务器等待客户端加载此地图，客户端如果加载成功，会发送Join信息到服务器  
5.如果接受连接，服务器将调用 AGameMode::Login该函数的作用是创建一个PlayerController，可用于在今后复制到新连接的客户端。成功接收后，这个PlayerController 将替代客户端的临时PlayerController （之前被用作连接过程中的占位符）。  
此时将调用 APlayerController::BeginPlay。应当注意的是，在此 actor 上调用RPC 函数尚存在安全风险。您应当等待 AGameMode::PostLogin 被调用完成。  
6.如果一切顺利，AGameMode::PostLogin 将被调用。这时，可以放心的让服务器在此 PlayerController 上开始调用RPC 函数。

那么这里面第5点需要重点强调一下。我们知道所谓连接，不过就是客户端连接到一个服务器，在维持着这个连接的条件下，我们才能真正的玩“网络游戏”。通常，如果我们想让服务器把某些特定的信息发送给特定的客户端，我们就需要找到服务器与客户端之间的这个连接。这个链接的信息就存储在PlayerController的里面，而这个PlayerController不能是随随便便创建的PlayerController，一定是客户端第一次链接到服务器，服务器同步过来的这个PlayerController（也就是上面的第五点，后面称其为**拥有连接的PlayerController**）。进一步来说，这个Controller里面包含着相关的NetDriver，Connection以及Session信息。

对于任何一个Actor（客户端上），他可以有连接，也可以无连接。一旦Actor有连接，他的Role（控制权限）就是ROLE\_AutonomousProxy，如果没有连接，他的Role（控制权限）就是ROLE\_SimulatedProxy 。

那么对于一个Actor，他有三种方法来得到这个连接（或者说让自己属于这个连接）。

1. **设置自己的owner为拥有连接的PlayerController，或者自己owner的owner为拥有连接的PlayerController。**也就说官方文档说的查找他最外层的owner是否是PlayerController而且这个PlayerController拥有连接。
2. **这个Actor必须是Pawn并且Possess了拥有连接的PlayerController。**这个例子就是我们打开例子程序时，开始控制一个角色的情况。我们控制的这个角色就拥有这个连接。
3. **这个Actor设置自己的owner为拥有连接的Pawn。**这个区别于第一点的就是，Pawn与Controller的绑定方式不是通过Owner这个属性。而是Pawn本身就拥有Controller这个属性。所以Pawn的Owner可能为空。（Owner这个属性在Actor里面，蓝图也可以通过GetOwner来获取）

对于组件来说，那就是先获取到他所归属的那个Actor，然后再通过上面的条件来判断。

我这里举几个例子，玩家PlayerState的owner就是拥有连接的PlayerController，Hud的owner是拥有连接的PlayerController，CameraActor的owner也是拥有连接的PlayerController。而客户端上的其他NPC（一定是在服务器创建的）是都没有owner的Actor，所以这些NPC都是没有连接的，他们的Role就为ROLE\_SimulatedProxy。

所以我们发现这些与客户端玩家控制息息相关的Actor才拥有所谓的连接。不过，进一步来讲，我们要这连接还有什么用？好吧，照搬官方文档。

连接所有权是以下情形中的重要因素：  
1.RPC 需要确定哪个客户端将执行运行于客户端的 RPC  
2.Actor 复制与连接相关性  
3.在涉及所有者时的 Actor 属性复制条件

对于RPC，我们知道，UE4里面在Actor上调用RPC函数，可以实现类似在客户端与服务器之间发送可执行的函数的功能。最基本的，当我一个客户端拥有ROLE\_AutonomousProxy权限的Actor在服务器代码里调用RPC函数（UFUNCTION(Reliable, Client)）时，我怎么知道应该去众多的客户端的哪一个里面执行这个函数。（RPC的用法不细说，参考官方文档）答案就是通过这个Actor所包含的连接。关于RPC进一步的内容，下个问题里再详细描述。

第二点，Actor本身是可以同步的，他的属性当然也是。这与连接所有权也是息息相关。因为有的东西我们只需要同步给特定的客户端，其他的客户端不需要知道，（比如我当前的摄像机相关内容）。

对于第三点，其实就是Actor的属性是否同步可以进一步根据条件来做限制，有时候我们想限制某个属性只在拥有ROLE\_AutonomousProxy的Actor使用，那么我们对这个Actor的属性ReplicatedMovement写成下面的格式就可以了。

void AActor::GetLifetimeReplicatedProps( TArray< FLifetimeProperty > & OutLifetimeProps ) const

{

DOREPLIFETIME\_CONDITION( AActor, ReplicatedMovement, COND\_AutonomousOnly );

}

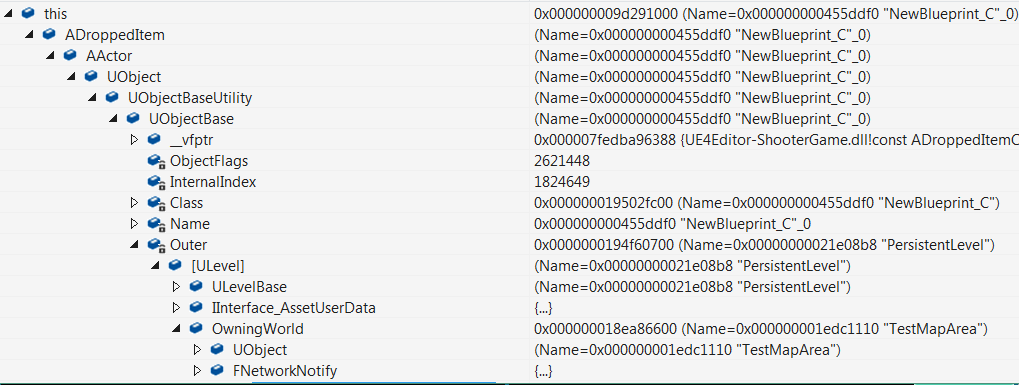
而经过前面的讨论我们知道ROLE\_AutonomousProxy与所属连接是密不可分的。

最后，这里留一个思考问题：**如果我在客户端创建出一个Actor，然后把它的Owner设置为带连接的PlayerController，那么他也有连接么？**这个问题在下面的一节中回答。

**1.Actor的Role是ROLE\_Authority就是服务端么？**

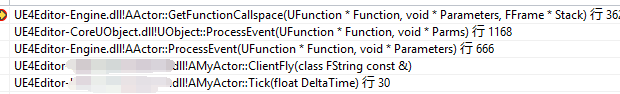
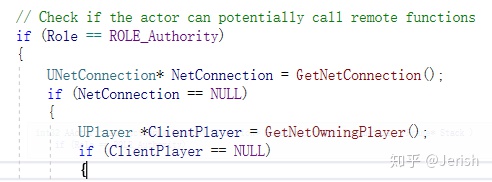
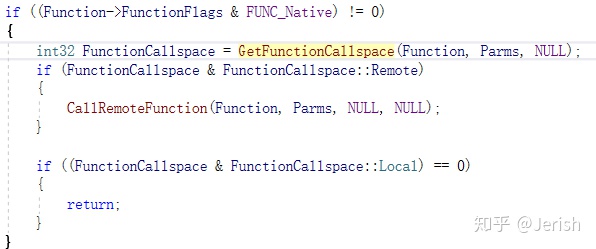
并不是，有了前面的讲述，我们已经可以理解，如果我在客户端创建一个独有的Actor(不能勾选bReplicate）。那么这个Actor的Role就是ROLE\_Authority，所以这时候你就不能通过判断他的Role来确定当前调试的是客户端还是服务器。这时候最准确的办法是获取到NetDiver，然后通过NetDiver找到Connection。（事实上，GetNetMode()函数就是通过这个方法来判断当前是否是服务器的）对于服务器来说，他只有N个ClientConnections，对于客户端来说只有一个serverConnection。

如何找到NetDriver呢?可以参考下面的图片，从Outer获取到当前的Level，然后通过Level找到World。World里面就有一个NetDiver。当然，方法不止这一个了，如果有Playercontroller的话，Playercontroller上面也有NetConnection，可以再通过NetConnection再获取到NetDiver。



**2.Owner是如何在RPC调用中生效的？**

答案在AActor::GetFunctionCallspace里面，每次调用RPC函数时，会调用该函数判断当前是不是在远端调用，是的会就会通过网络发送RPC。GetFunctionCallspace里面会通过Owner找到connection信息。



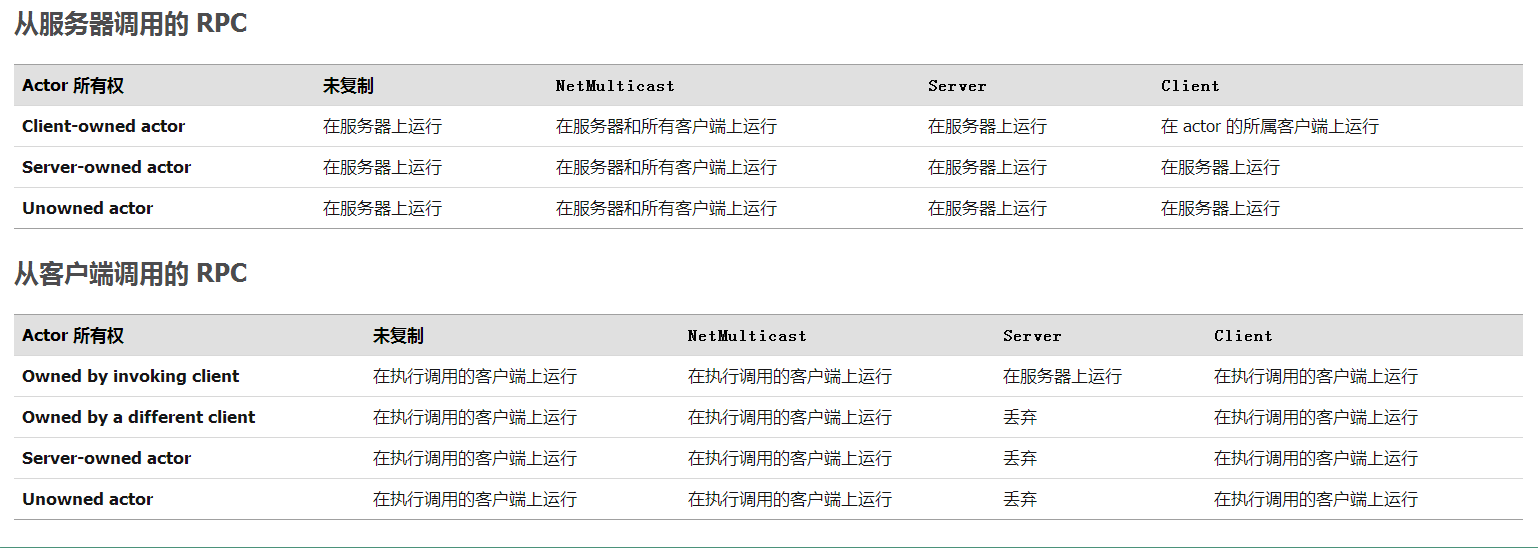
**二. 进一步理解RPC与同步**

**1. RPC函数应该在哪个端执行？**

**对于一个形如UFUNCTION(Reliable, Client)的RPC函数，我们知道这个函数应该在服务器调用，在客户端执行。可是如果我在Standalone的端上执行该函数的时候会发生什么呢？**

答案是在服务器上执行。其实这个结果完全可以参考下面的这个官方图片。

刚接触RPC的朋友可能只是简单的记住这个函数应该从哪里调用，然后在哪里执行。不过要知道，即使我声明一个在服务器调用的RPC我还是可以不按套路的在客户端去调用（有的时候并不是我们故意的，而是编写者没有理解透彻），其实这种不合理的情况UE早就帮我想到并且处理了。比如说你让自己客户端上的其他玩家去调用一个通知服务器来执行的RPC，这肯定是不合理的，因为这意味着你可以假装其他客户端随意给服务器发消息，这种操作与作弊没有区别~所以RPC机制就会果断丢弃这个操作。



所以大家可以仔细去看看上面的这个图片，对照着理解一下各个情况的执行结果，无非就是三个变量：

1. 在哪个端调用
2. 当前执行RPC的Actor归属于哪个连接
3. RPC的类型是什么。

**2. 客户端创建的Actor能调用RPC么？**

不过看到这里，再结合上一节结尾提到的问题，如果我在客户端创建一个Actor。把这个Actor的Owner设置为一个带连接PlayerController会怎么样呢？如果在这里调用RPC呢？

我们确实可以通过下面这种方式在客户端给新生成的Actor指定一个Owner。



好吧，关键时候还是得搬出来官方文档的内容。

您必须满足一些要求才能充分发挥 RPC 的作用：  
1. 它们必须从 Actor 上调用。  
2. Actor 必须被复制。  
3. 如果 RPC 是从服务器调用并在客户端上执行，则只有实际拥有这个 Actor 的客户端才会执行函数。  
4. 如果 RPC 是从客户端调用并在服务器上执行，客户端就必须拥有调用 RPC 的 Actor。 5. 多播 RPC 则是个例外：  
o 如果它们是从服务器调用，服务器将在本地和所有已连接的客户端上执行它们。  
o 如果它们是从客户端调用，则只在本地而非服务器上执行。  
o 现在，我们有了一个简单的多播事件限制机制：在特定 Actor 的网络更新期内，多播函数将不会复制两次以上。按长期计划，我们会对此进行改善，同时更好的支持跨通道流量管理与限制。

看完第二条，其实你就能理解了，你的Actor必须要被复制，也就是说必须是bReplicate属性为true， Actor是从服务器创建并同步给客户端的（客户端如果勾选了bReplicate就无法在客户端上正常创建，参考第4部分）。

所以，这时候调用RPC是失效的。我们不妨去思考一下，连接存在的意义本身就是一个客户端到服务器的关联，这个关联的主要目的就是为了执行同步。如果我只是在客户端创建一个给自己看的Actor，根本就不需要网络的连接信息（当然你也没有权限把它同步给服务器），所以就算他符合连接的条件，仍然是一个**没有意义的连接**。

同时，我们可以进一步观察这个Actor的属性，除了Role以外，Actor身上还有一个RemoteRole来表示他的对应端（如果当前端是客户端，对应端就是服务器，当前端是服务器，对应端就是客户端）。你会发现这个在客户端创建的Actor，他的Role是ROLE\_Authority（并不是ROLE\_AutonomousProxy），而他的RemoteRole是ROLE\_None。这也说明了，这个Actor只存在于当前的客户端内。

**3. RPC与Actor同步谁先执行？**

下面我们讨论一下RPC与同步直接的关系，这里提出一个这样的问题

**问题**：**服务器ActorA在创建一个新的ActorB的函数里同时执行自身的一个Client的RPC函数，RPC与ActorB的同步哪个先执行？(原答案不准确已修改)**

答案是不确定。这个涉及到UE4网络消息的发送机制与发送时机，一般来说，RPC的数据会立刻塞到SendBuffer里面，而Actor的同步要等到NetDriver统一处理。所以RPC的消息是相对靠前的，不过由于存在丢包延迟等情况，这个结果在网络环境下不能确定。

**那么这个问题会造成什么后果呢？**

1. 当你创建一个新的Actor的同时（比如在一个函数内），你将这个Actor作为RPC的参数传到客户端去执行，这时候你会发现客户端的RPC函数的参数为NULL。
2. 你设置了一个bool类型属性A并用UProperty标记了一个回调函数OnRep\_Use。你先在服务器里面修改了A为true，同时你调用了一个RPC函数让客户端把A置为true。结果就导致你的OnRep\_Use函数没有执行。但实际上，这会导致你的OnRep\_Use函数里面还有其他的操作没有执行。

如果你觉得上面的情况从来没有出现过，那很好，说明暂时你的代码没有类似的问题，但是我觉得有必要提醒一下大家，因为UE4代码里面本身就有这样的问题，你以后也很有可能遇到。下面举例说明实际可能出现的问题：

**情况1：**当我在服务器创建一个NPC的时候，我想让我的角色去骑在NPC上并控制这个NPC，所以我立刻就让我的Controller去Possess这个NPC。在这个过程中，PlayerController就会执行

UFUNCTION(Reliable, Client)

void ClientRestart (APawn\*NewPawn)

当客户端收到这个RPC函数回调的时候就发现我的APlayerController::ClientRestart\_Implementation (APawn\* NewPawn)里面的参数为空~原因就是因为这个NPC刚在服务器创建还没有同步过来。同理，对于刚出生的玩家来说是不是也存在这个问题呢？

确实存在。不过，UE4本身其实有考虑到这个问题，ClientRestart函数有特殊的处理，服务器在移动的时候会通过SafeRetryClientRestart让客户端执行ClientRetryClientRestart，如果发现Pawn不对（也就是第一次ClientRestart执行失败的话）就会触发再次执行。

更新：我们可以通过设置控制台变量net.DelayUnmappedRPCs 1允许客户端等到这个对象生成的时候再去执行，但是仅限于可靠的RPC。

**情况2：**对于Pawn里面的Controller成员声明如下

UPROPERTY(replicatedUsing=OnRep\_Controller)

AController\*Controller;

OnRep\_Controller回调函数里面回去执行Controller->SetPawnFromRep(this);

进而执行

Pawn = InPawn;

OnRep\_Pawn();

下面重点来了，OnRep\_Pawn函数里面会执行

OldPawn->Controller= NULL;

将客户端之前Controller控制的角色的Controller设置为空。

到现在来看没有什么问题。那么现在结合上面第二个问题，如果一个RPC函数的执行发生在客户端的Controller同步前同时修改为正确的Controller，那么OnRep\_Controller回调函数就不会执行。所以客户端的原来Controller控制的OldPawn的Controller就不会置为空，导致的结果是客户端和服务器竟然不一样。

实际上，确实存在这么一个函数，这个RPC函数就是ClientRestart。这看起来就很奇怪，因为ClientRestart如果没有正常执行的话，OnRep\_Controller就会执行，进而导致客户端的oldPawn的Controller为空（与服务器不同，因为服务器并没有去设置OldPawn的Controller）。不知道这是不是引擎的一个bug。

不管怎么说，你需要清楚的是**RPC的执行与同步的执行是有先后关系的，而这种关系会影响到代码的逻辑**，所以之后的代码有必要考虑到这一点。

最后，对使用RPC的朋友做一个提醒，有些时候我们在使用UPROPERTY标记Server的函数时，可能是从客户端调用，也可能是从服务器调用。虽然结果都是在服务器执行，但是过程可完全不同。从客户端调用的在实际运行时是通过网络来处理的，一定会有延迟。而从服务器调用的则会立刻执行。

**4. 多播MultiCast RPC会发送给所有客户端么？（已修改）**

**原答案：**

看到这个问题，你可能想这还用说么？不发给所有客户端那要多播干什么？但事实上确实不一定。

考虑到服务器上的一个NPC，在地图的最北面，有两个客户端玩家。一个玩家A在这个NPC附近，另一个玩家B在最南边看不到这个NPC（实际上就是由于距离太远，服务器没有把这个Actor同步到这个B玩家的客户端）。我们现在在这个NPC上调用多播RPC通知所有客户端上显示一个提示消失“NPC发现了宝藏”。这个消息会不会发送到B客户端上面？

* 情况一：会。多播顾名思义就是通知所有客户端，不需要考虑发送到哪一个客户端，直接遍历所有的连接发送即可。
* 情况二：不会。RPC本来就是基于Actor的，在客户端B上面连这个Actor都没有，我还可以使用RPC不会很奇怪？

第一种情况强化了多播的概念，淡化了RPC基于Actor的机制，情况二则相反。所以看起来都有道理。实际上，UE4里面更偏向第二种情况，处理如下：

如果一个多播标记为Reliable，那么他默认会给所有的客户端执行该多播事件，如果其标记的是unreliable，他就会检测该NPC与客户端B的网络相关性（即在客户端B上是否同步）。但实际上，UE还是认为开发者不应该声明一个Reliable的多播函数。下面给出UE针对这个问题的相关注释：（相关的细节在另一篇进一步深入UE网络同步的文章里面去分析）

// Do relevancy check if unreliable.

// Reliables will always go out. This is odd behavior. On one hand we wish to garuntee "reliables always getthere". On the other

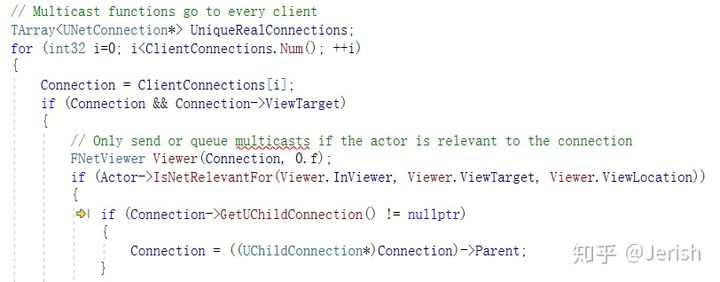
// hand, replicating a reliable to something on theother side of the map that is non relevant seems weird.

// Multicast reliables should probably never beused in gameplay code for actors that have relevancy checks. If they are, the

// rpc will go through and the channel will be closedsoon after due to relevancy failing.

**修改答案：**

我发现引擎在这块已经更改了，引擎已经不考虑多播函数是否是reliable了，只要不满足NetRelevant都不会发送，参考UNetDriver::ProcessRemoteFunction。



**5. RPC参数与返回值**

**参数：**RPC函数除了UObject类型的指针以及constFString&的字符串外，其他类型的指针或者引用都不可以作为RPC的参数。对于UObject指针类型我们可以在另一端通过GUID识别（后面第五部分有讲解），但是其他类型的指针传过去是什么呢？我们根本就无法还原其地址，所以不允许传输其指针或者引用。

而对于FString，传const原因我认为是为了不想让发送方与接收方两边对字符串进行修改，而传引用只是为了减少复制构造带来的开销。在FString发送与接收的处理细节里面并不在意其是否是const&，他只在意他的类型以及相对Object的偏移。

**返回值：**一个RPC函数是不能有返回值的，因为其本身的执行就是一次消息的传递。假如一个客户端执行一个Server RPC，如果有返回值的话，那么岂不是服务器执行后还要再发送一个消息给客户端？这个消息怎么处理？再发一次RPC？如果还有返回值那么不就无限循环了？因此RPC函数不可以添加返回值。

**三. 合理使用COND\_InitialOnly**

前面提到过，Actor的属性同步可以通过这种方式来实现。

声明一个属性并标记

UPROPERTY(Replicated)

uint8 bWeapon: 1;

UPROPERTY(Replicated)

uint8 bIsTargeting: 1;

**void** Character**::**GetLifetimeReplicatedProps(TArray**<**FLifetimeProperty **>** **&** OutLifetimeProps ) **const**

{

DOREPLIFETIME(Character,bWeapon );

DOREPLIFETIME\_CONDITION(Character, bIsTargeting, COND\_InitialOnly

);

这里面的第一个属性一般的属性复制，第二个就是条件属性复制。条件属性复制无非就是告诉引擎，这个属性在哪些情况下同步，哪些情况下不同步。这些条件都是引擎事先提供好的。

这里我想着重的提一下COND\_InitialOnly这个条件宏，汉语的官方文档是这样描述的：该属性仅在初始数据组尝试发送。而英文是这样描述的：This property will only attempt to send on the initial bunch。对比一下，果然还是英文看起来更直观一点。

经过测试，这个条件的效果就是这个宏声明的属性只会在Actor初始化的时候同步一次，接下来的游戏过程中不会再同步。所以，我们大概能想到这个东西在有些时候确实用的到，比如同步玩家的姓名，是男还是女等，这些游戏开始到结束一般都不会改变的属性。也就是说，上限一般调整的次数很少，如果真的有调整并需要同步，他会手动调用函数去同步该属性。这样就可以减少同步带来的压力。 然而，一旦你声明为COND\_InitialOnly。你就要清楚，同步只会执行一次，客户端的OnRep回调函数就会执行一次。所以，当你在服务器创建了一个新的Actor的时候你需要第一时间把需要改变的值修改好，一旦你在下一帧（或是下一秒）去执行那么这个属性就无法正确的同步到客户端了。

**四．客户端与服务器一致么？**

我们已经知道UE4的客户端与服务器公用一套代码，那么我们在每次写代码的时候就有必要提醒一下自己。这段代码在哪个端执行，客户端与服务器执行与表现是否一致？

虽然，我很早之前就知道这个问题，但是写代码的时候还是总是忽略这个问题，而且程序功能经常看起来运行的没什么问题。不过看起来正常不代表逻辑正常，有的时候同步机制帮你同步一些东西，有时候会删除一些东西，有时候又会生成一些东西，然而你可能一点都没发现。

举个例子，我在一个ActorBeginPlay的时候给他创建一个粒子Emiter。代码大概如下：

void AGate::BeginPlay()

{

Super::BeginPlay();

//单纯的在当前位置创建粒子发射器

GetWorld()->SpawnActor<AEmitter>(SpawnEmitter,GetActorLocation(), UVictoryCore::RTransform(SpawnEmitterRotationOffset,GetActorRotation()));

}

代码很简单，不过也值得我们分析一下。

首先，服务器下，当Actor创建的时候就会执行BeginPlay，然后在服务器创建了一个粒子发射器。这一步在服务器（DedicateServer）创建的粒子其实就是不需要的，所以一般来说，这种纯客户端表现的内容我们不需要在专用服务器上创建。

再来看一下客户端，当创建一个Gate的时候，服务器会同步到客户端一个Gate，然后客户端的Gate执行BeginPlay，创建粒子。这时候我们已经发现二者执行BeginPlay的时机不一样了。进一步测试，发现当玩家远离Gate的时候，由于UE的同步机制（只会同步一定范围内的Actor），客户端的Gate会被销毁，而粒子发射器也会销毁。而当玩家再次靠近的时候，Gate又被同步过来了，原来的粒子发射器也被同步过来。而因为客户端再次执行了BeginPlay，又创建了一个新的粒子，这样就会导致不断的创建新的粒子。

你觉得上面的描述准确么？

并不准确，因为上述逻辑的执行还需要一个前置条件——这个粒子的bReplicate属性是为false的。有的时候，我们可能一不小心就写出来上面这种代码，但是表现上确实正常的，为什么？因为SpawnActor是否成功是有条件限制的，在生成过程中有一个函数

bool AActor::TemplateAllowActorSpawn(UWorld\* World,const FVector& AtLocation, const FRotator& AtRotation, const struct

FActorSpawnParameters& SpawnParameters)

{

return !bReplicates || SpawnParameters.bRemoteOwned||World->GetNetMode() != NM\_Client;

}

如果你是在客户端，且这个Actor勾选了bReplicate的话，TemplateAllowActorSpawn就会返回false，创建Actor就会失败。如果这个Actor没有勾选bReplicate的话，那么服务器只会创建一个，客户端就可能不断的创建，而且服务器上的这个Actor与客户端的Actor没有任何关系。

另外，还有一种常见的错误。就是我们的代码执行是有条件的，然而这个条件在客户端与服务器是不一样的（没同步）。如

void Gate::CreateParticle(int32 ID)

{

if(GateID!= ID)

{

FActorSpawnParameters SpawnInfo;

GetWorld()->SpawnActor<AEmitter>(SpawnEmitter, GetActorLocation(),GetActorRotation(), SpawnInfo);

}

}

这个GateID是我们在GateBeginPlay的时候随机初始化的，然而这个GateID只在服务器与客户端是不同的。所以需要服务器同步到客户端，才能按照我们理想的逻辑去执行。

**五. 属性同步的基本规则与注意事项**

非休眠状态下的Actor的属性同步：只在服务器属性值发生改变的情况下执行  
回调函数执行条件：服务器同步过来的数值与客户端不同  
休眠的Actor：不同步

首先要认识到，同步操作触发是由服务器决定的，所以不管客户端是什么值，服务器觉得该同步就会把数据同步到客户端。而回调操作是客户端执行，所以客户端会判断与当前的值是否相同来决定是否产生回调。

然后是属性同步，属性同步的基本原理就是服务器在创建同步通道的时候给每一个Actor对象创建一个属性变化表（这里面涉及到FObjectReplicator，FRepLayout，FRepState，FRepChangedPropertyTracker相关的类，有兴趣可以进一步了解，在另一深入UE网络同步文章里有讲解），里面会记录一个当前默认的Actor属性值。之后，每次属性发生变化的时候，服务器都会判断新的值与当前属性变化表里面的值是否相同，如果不同就把数据同步到客户端并修改属性变化表里的数据。对于一个非休眠且保持连接的Actor，他的属性变化表是一直存在的，所以他的表现出来的同步规则也很简单，只要服务器变化就同步。

动态数组TArray在网络中是可以正常同步的，系统会检测到你的数组长度是否发生了变化，并通知客户端改变。

**1. 结构体的属性同步**

注意，UE里面UStruct类型的结构体在反射系统中对应的是UScriptStruct，他本身可以被标记Replicated并且结构体内的数据默认都会被同步，而且如果里面有还子结构体的话也仍然会递归的进行同步。如果不想同步的话，需要在对应的属性标记NotReplicated，而且这个标记只对UStruct有效，对UClass无效。

有一点特别的是，Struct结构内的数据是不能标记Replicated的。如果你给Struct里面的属性标记replicated，UHT在编译的时候就会提醒你编译失败。

最后，UE里面的UStruct不可以以成员指针的方式在类中声明。

**2. 属性回调**

**问题：属性回调与RPC在使用结果上的差异？**

属性回调理论上一定会执行，而RPC函数有可能由于错过执行时机而不再会执行。例如：我在服务器上面有一个宝箱，第一个玩家过去后，宝箱会自动开启。如果使用RPC函数，当第一个玩家过去后，箱子执行多播RPC函数触发开箱子操作。但是由于其他的玩家离这个箱子很远，所有这个箱子没有同步给其他玩家，其他玩家收不到这个RPC消息。（如果对结果有疑问参考第二节的第四个问题）当这些玩家之后再过去之后，会发现箱子还是关闭的。如果采用属性回调，但第一个玩家过去后，设置箱子的属性bOpen为true，然后同步到所有客户端，通过属性回调执行开箱子操作。这时候其他玩家靠近箱子时，箱子会同步到靠近的玩家，然后玩家在客户端上会收到属性bOpen，同时执行属性回调，这时候可以实现所有靠近的玩家都会发现箱子已经被别人开过了。

**问题：服务器上生成一个Actor，他在客户端上的UObject类型指针的属性回调与他的Beginplay谁先执行？**

这个问题这么看有点奇怪，我进一步描述一下。有一个类MyActor，他有一个指针属性PropertyB指向一个同步的MyActorB，同时这个指针属性有一个回调函数。现在我在服务器创建一个新的MyActor A，并设置A的PropertyB为MyActorB。那么在客户端上，是A的BeginPlay先执行，还是PropertyB的属性回调先执行？

答案是不确定，一开始的时候，我一直认为是属性回调在Actor的BeginPlay之前执行，测试了很多次也是这样的。但是某种情况下， BeginPlay会先执行。这个问题的意义就在于，一个Actor同步过去执行BeginPlay的时候，你发现他的属性还没有同步过来（而且只发现指针可能没有同步过来，其他内置类型都会在BeginPlay 前同步过来）。为什么指针没有同步过来？因为这个指针同步过来的时候，他指向的对象在客户端还不存在，他在客户端上也没有对应的GUID缓存 。由于找不到对应的对象，他只能先暂时记录下这个指针指向对象的GUID，然后在其他的Tick时间再回来检测这个对象是否存在。这种情况一般来说很难重现，不过这个问题有助于我们进一步加深对网络的理解。

**3. UObject指针类型的属性同步**

属性同步也好，RPC参数也好。我们都需要思考一下，我在传递一个UObject类型的指针时，这个UObject在客户端存在么？如果存在，我如何能通过服务器的一个指针找到客户端上相同UObject的指针？

答案是通过FNetworkGUID。服务器在同步一个对象引用（指针）的时候，会给其分配专门的FNetworkGUID并通过网络进行发送。客户端上通过识别这个ID，就可以找到对应UObject。

**那么如此说来，是不是只有标记Replicate的对象才能同步其引用或指针呢？**

也不是。对于直接从数据包加载出来的对象（如地图里面实现搭建好的建筑地形），我们可以直接认为服务器上的该地形对象与客户端上对应的地形对象就是一个对象，那么在服务器上指向该地形的指针发送到客户端也应该就是指向对应地形的指针。所以总结来说一个UObject对象是否可以通过网络发送他的引用有如下条件（参考官方文档）：

**您通常可以按照以下原则来确定是否可以通过网络引用一个对象：**  
任何复制的 actor 都可以复制为一个引用  
任何未复制的 actor 都必须有可靠命名（直接从数据包加载）  
任何复制的组件都可以复制为一个引用  
任何未复制的组件都必须有可靠命名。  
其他所有 UObject（非actor 或组件）必须由加载的数据包直接提供  
  
**什么是拥有可靠命名的对象？**  
拥有可靠命名的对象指的是存在于服务器和客户端上的同名对象。  
1.如果Actor 是从数据包直接加载(并非在游戏期间生成)，它们就被认为是拥有可靠命名。  
  
2.满足以下条件的组件即拥有可靠命名：  
● 从数据包直接加载  
● 通过construction scripts脚本添加  
● 采用手动标记（通过 UActorComponent::SetNetAddressable 设置）  
● 只有当您知道要手动命名组件以便其在服务器和客户端上具有相同名称时，才应当使用这种方法（最好的例子就是 AActor C++ 构造函数中添加的组件）

最后总结一下就是有四种情况下UObject对象的引用可以在网络上传递成功

1. 标记replicate
2. 从数据包直接Load
3. 通过Construction scripts添加或者C++构造函数里面添加
4. 使用UActorComponent::SetNetAddressable标记（这个只针对组件，其实蓝图里面创建的组件默认就会执行这个操作）

**六．组件同步**

组件在同步上分为两大类：**静态组件与动态组件**。

**对于静态组件：**一旦一个Actor被标记为同步，那么这个Actor身上默认所挂载的组件也会随Actor一起同步到客户端（也需要序列化发送）。什么是默认挂载的组件?就是C++构造函数里面创建的默认组件或者在蓝图里面添加构建的组件。所以，这个过程与该组件是否标记为Replicate是没有关系的。

**对于动态组件：**就是我们在游戏运行的时候，服务器创建或者删除的组件。比如，当玩家走进一个洞穴时，给洞穴里面的火把生成一个粒子特效组件，然后同步到客户端上，当玩家离开的时候再删除这个组件，玩家的客户端上也随之删除这个组件。

对于动态组件，我们必须要attach到Actor上并设置他的Replicate属性为true，即通过函数 AActorComponent ::SetIsReplicated(true)来操作。而对于静态组件，如果我们不想同步组件上面的属性，我们就没有必要设置Replicate属性。

一旦我们执行了SetIsReplicated(true)。那么组件在属性同步以及RPC上与Actor的同步几乎没有区别，组件上也需要设置GetLifetimeReplicatedProps来执行属性同步，Actor同步的时候会遍历他的子组件查看是否标记Replicate以及是否有属性要同步。

**注意：**动态组件的同步是有限制的。由于组件里面的很多成员是无法同步的（比如skeletalmesh）导致很多组件服务器创建后并不会显示在客户端上面，所以客户端在收到之后还要再进行一些相关的处理，比如说把组件放到一个属性并在回调里面重新加载一遍skeletalmesh（记录skeletalmesh的TSoftObjectPtr并同步，客户端收到后执行LoadSynchronous本地加载）。

bool AActor::ReplicateSubobjects(UActorChannel \*Channel, FOutBunch \*Bunch, FReplicationFlags \*RepFlags)

{

check(Channel);

check(Bunch);

check(RepFlags);

bool WroteSomething = false;

for(UActorComponent\* ActorComp : ReplicatedComponents)

{

if(ActorComp && ActorComp->GetIsReplicated())

{

//Lets the component add subobjects before replicating its own properties.

WroteSomething|= ActorComp->ReplicateSubobjects(Channel, Bunch,RepFlags);

//(this makes those subobjects 'supported', and from here on those objects mayhave reference replicated) 子对象（包括子组件）的同步，其实是在ActorChannel里进行

WroteSomething |= Channel->ReplicateSubobject(ActorComp,\*Bunch,\*RepFlags);

}

}

return WroteSomething;

}

对于C++默认的组件，需要放在构造函数里面构造并设置同步，UE给出了一个例子：

ACharacter::ACharacter()

{

// Etc...

CharacterMovement = CreateDefaultSubobject<UMovementComp\_Character>(TEXT("CharMoveComp");

if (CharacterMovement)

{

CharacterMovement->UpdatedComponent = CapsuleComponent;

CharacterMovement->GetNavAgentProperties()->bCanJump = true;

CharacterMovement->GetNavAgentProperties()->bCanWalk = true;

CharacterMovement->SetJumpAllowed(true);

//Make DSO components net addressable 实际上如果设置了Replicate之后，这句代码就没有必要执行了

CharacterMovement->SetNetAddressable();

// Enable replication by default

CharacterMovement->SetIsReplicated(true);

}

}

# 移动的网络同步

绝大部分游戏都需要移动功能，玩家控制角色需要响应玩家操控，野怪、NPC角色需要能自动行走，显得有智慧。对于网络游戏，则要通过服务器把移动同步给其他玩家，由于网络状况的不确定，还需处理网络抖动、丢包和延迟等情况。可见移动功能是游戏中最基础但相当复杂的功能。UE4引擎经过多年发展，移动网络同步功能已比较完善，本文将简要介绍UE4如何通过移动组件，来完成网络同步功能的。

UE4引擎代码开源，可从Epic launcher或github上下载到相应源码，以下部分内容会具体到源码中的特定函数或属性，可以参照源码进行查看。

**Actor和移动组件Component**

对于还不太熟悉UE4引擎的读者，可以先了解下UE4中的一些基本概念。

Actor代表UE4中的“实体”，通常是可见的，比如玩家、武器、车辆都是Actor，它们在场景中有位置信息。

UE4充分使用了组合模式，在描述Actor移动时，并不是“某些种类Actor可以移动”，而是“具有移动组件的Actor都可以移动”。 移动组件为UMovementComponent，它是基类，提供基本的移动功能，有多个不同功能的子类，一些子类移动组件专门用于服务一种特殊的Actor，比如通常代表玩家的ACharacter。

UNavMovementComponent

用于处理AI控制实体的移动，需要做自动寻路方面的支持，让实体移动更加自然。

UPawnMovementComponent

继承自UNavMovementComponent，用于处理Pawn的移动，并且能通过AddInputVector()等接口接受输入。和APawn一起使用。

UCharacterMovementComponent

最复杂的一个移动组件。支持walking, falling, flying等多种移动模式，支持网络同步(其他移动组件都不支持)，支持移动平滑、预测，并且处理了上坡、撞墙、台阶等多种情况，可以认为是人型角色移动的最佳选择了。和ACharacter一起使用。

**UpdateComponent**

Actor本身没有位置概念，其位置属性由SceneComponent赋予，Actor的RootComponent就是一个SceneComponent。因此，理论上移动组件不是改变某个Actor的位置，而是改变Actor上某个SceneComponent的位置，通常就是RootComponent，当RootComponent位置变了，Actor位置自然就变了。UMovementComponent的UpdatedComponent属性就是当前组件改变位置的Component，可动态设置。

**UCharacterMovement**

UCharacterMovement最为复杂，而且UE4移动同步功能在其中实现，因此主要分析它。

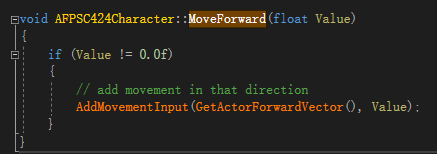
**玩家输入**

作为一个玩家控制角色，移动的第一步就是接受输入，比如按键和鼠标移动，这样角色才知道往哪移动，视角往哪看。

APawn上有SetupPlayerInputComponent(UInputComponent\* InputComponent)接口，可以在里面对不同输入绑定对应的响应函数，以MoveForward为例：



该Character向"MoveForward"输入事件绑定了MoveForward函数，函数内容也很简单：



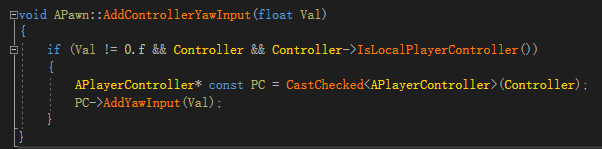
最终调用到了UPawnMovementComponent::AddInputVector函数，提供了一个向前的向量。

那么MoveForward函数何时被触发？在APlayerController执行TickActor函数时，会用PlayerInput检查玩家提供的输入，调用绑定的函数。比如"W"键绑定了"MoveForward"事件，那么玩家只要按下"W"键，每帧Tick都会触发MoveForward函数，Value固定为1。至于其他方向的移动，处理方式与MoveForward类似，都使用了AddMovementInput接口，只是向量方向不同。

转向的实现和移动有些不同，通常有两种方式。

第一种是FPS游戏常用的，让角色转向与视角转向一致，这可以不依赖移动组件，由PlayerController实现。AController有ControlRotation属性，表示当前玩家的朝向，APlayerController有RotationInput属性，表示玩家的转向输入。在射击游戏中，我们通常希望**视角**能立即响应旋转输入，即使控制的是相对笨重，移动缓慢的角色。视角转动后，让角色**自身朝向**实时与视角同步，或者角色花费一定时间转动到视角朝向，都是合理的。因此转向其实可以分为两部分：视角转向和角色转向。我们可以完全让PlayerController接受转向输入，把PlayerController朝向作为视角朝向，然后让角色以一定策略跟随PlayerController的转向。

在FPS游戏中，可以在收到玩家转向输入时直接修改Controller的转向，然后Character设置bUseControllerRotationYaw为true，Yaw朝向完全与Controller同步，玩家自身的Roll和Pitch都不变：



第二种做法适用第三人称视角或自由视角游戏，可以让玩家转向与移动输入造成的加速度方向一致，与玩家视角无关。通过把bOrientRotationToMovement属性设为false实现，这个配置与bUseControllerRotationYaw等配置冲突。



还有更复杂的转向操作，比如操控坦克，车身、炮塔可以单独转动，这就需要一些特殊处理了。



**网络同步**

我们常用的多人游戏框架是Client/DedicatedServer模式，有一个专门的服务器，连接服务器的玩家称为客户端。玩家控制的角色Actor存在于三个地方：玩家本地、服务器、其他客户端，相应的，角色的NetRole分别为Autonomous，Authority和Simulate，仅Autonomous是本地接受玩家输入的角色。UE使用了属性同步模式，玩家必须把数据发往服务器，再由服务器统一把数据同步给其他客户端，因此这里讨论的是“属性同步模式下的移动同步”，帧同步技术不考虑。网络同步要做的事情，就是把玩家本地做的角色移动，通过DedicatedServer，同步到其他客户端上，这个过程中要体现一个良好的表现，需要满足几个要求：

* Autonomous客户端无延迟

这点很好理解，玩家希望操控的角色能立即响应移动输入，如果在其中引入网络延迟，将很容易察觉，且不可忍受

* 服务端位置是权威的

服务器上玩家位置肯定是最准的，所有客户端的位置必须要与其一致，包括Autonomous客户端，这就对Autonomous客户端提出更高要求了，需要在实时响应玩家输入的同时与服务器保持同步

* Simulate客户端上移动表现平滑

玩家位置每帧都要更新，但Simulate客户端不可能每帧收到位置更新信息，因此需要做适当的插值，让移动变得平滑

* 反外挂

其实就是服务器做位置更新检查，对于Autonomous发送的违法移动，比如飞天，予以拒绝

在这几个基础要求上，还可以有一些优化，比如：

* 移动同步流量优化

网络同步一个前提是UpdatedComponent没有开启物理模拟，还是由移动组件控制位置，下面讨论的移动都是非物理模拟的。

网络同步情况下移动大概流程：

* 每一帧执行TickComponent时，会计算这一帧的加速度和转向，之后对于Autonomous类型character，会调用ReplicateMoveToServer函数
* ReplicateMoveToServer会把移动保存到PendingMove列表，再执行PerformMovement，然后调用ServerMove把移动同步给服务器，传给服务器的包括移动所用参数、客户端自己移动后的位置，以及当前时间戳
* ServerMove在服务器上执行，它解析移动参数，然后产生对应的移动。然后它查看结果位置，并计算自最后一次客户端响应以来的时间， 以及客户端声明的位置与服务器确定的位置之间的差异。如果差异足够大，服务器调用“ClientAdjustPosition”， 它将发送到Autonomous客户端并传递校正的位置。
* ClientAdjustPosition在客户端上执行，客户端把自己位置设置为服务器上的位置，然后设置bUpdatePosition标记位为true
* 当客户端再次调用“TickComponent”时，如果收到来自服务器的校正，客户端将在调用“PerformMovement”之前调用“ClientUpdatePosition” 。此过程将重演在服务器调整移动的时间戳之后发生的待定移动列表中的所有移动。

**为什么是UDP**

游戏开发中，通常使用UDP作为传输协议，因为游戏数据是时间敏感的，数据包需要尽快送达。而TCP是保证有序可达的，数据流中有一个数据包丢失，就要等它重传完成后，才能传输后续数据，称为head of blocking。但UDP不保证可达和有序，因此游戏中很多网络代码都在尽量处理这个问题。

**Autonomous角色移动**

Autonomous角色接受本地玩家控制，大部分移动逻辑与单机角色相同，最终都会执行PerformMovement真正执行移动，额外操作为把移动数据发往服务器。

移动组件的TickComponent中，首先依然会获取并清空玩家输入，然后处理跳跃状态，计算输入产生的加速度，这些和单机模式相同。之后会调用ReplicateMoveToServer函数，里面包括了移动数据发送和本地PeformMovement。

首先介绍一个重要数据结构：**FSavedMove\_Character**。它用于描述玩家的一次移动，发到服务器的移动数据来源于它，也可以用于位置更正和重放，可以认为它是一次移动的快照。

主要属性有：

TimeStamp：这次移动发生的时间

DeltaTime：这次移动使用的时间

CustomTimeDilation：时间膨胀系数，可以用于快进和慢放

StartPackedMovementMode：移动发生前的MovementMode

StartLocation：移动发生前的位置

StartVelocity：移动发生前的速度

EndPackedMovementMode：移动发生后的MovementMode

SavedLocation：移动发生后的位置

SavedVelocity：移动发生后的速度

Acceleration：移动所用加速度

其包含属性远比这里列举的多，可见其功能完善。理论上只要有这些数据，就能复盘整个移动过程，UE4的回放功能也确实是这么做的。

回到Autonomous移动，先简单介绍一下Autonomous移动与服务端同步问题。本地每次移动都会生成FSavedMove\_Character，并且维护一个SavedMoves数组，保存了当前玩家本地已经做的移动，但还没被服务器检查认可，如果服务器认可了一些移动，就把它们从SavedMoves里删掉，如果检查不通过，就执行异常处理流程。因此整个过程类似TCP传输的发送/确认机制，UE也确实用了很多"Ack"字眼，可见移动同步远比“把移动数据发给服务器，再同步给其他客户端”复杂。

**ReplicateMoveToServer**

首先会从SavedMoves里找到最早发生的一个"ImportantMove"。所谓"ImportantMove"，就是与最新被服务器Ack的Move有显著差异的Move（以下所说“Move”都是FSavedMove\_Character），两者MovementMode不同，跳跃、下蹲标记为不同、加速度改变明显，都会把这个未被Ack的Move作为"ImportantMove"。

之后，创建一个FSavedMove\_Character，并对它初始化，设置相应的信息。然后执行通用的PerformMovement，对角色执行真正的移动，处理碰撞、滑动等一系列复杂操作。移动结束后，就可以把Move中剩余属性补充上了，包括移动后的位置、转向、速度等，并且根据移动前后状况，会设置是否能被合并的标记，关于移动合并，会在之后介绍。

目前Move数据已填充完毕，可以把它加到SavedMoves数组中。

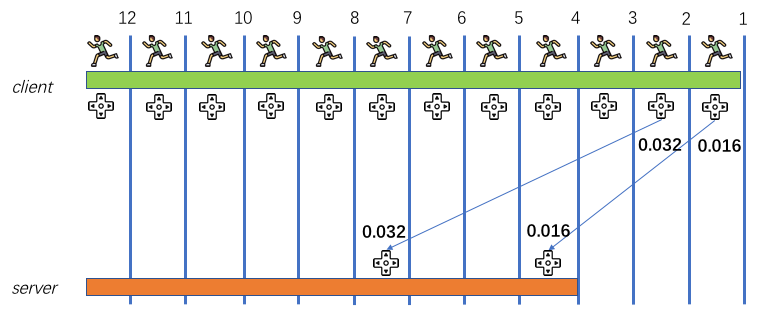
**移动时间戳**

观察FNetworkPredictionData\_Client\_Character和FSavedMove\_Character，会发现多个时间相关属性，它们有着不同的含义。

FNetworkPredictionData\_Client\_Character有ClientUpdateTime、CurrentTimeStamp、LastReceivedAckRealTime三个属性，FSavedMove\_Character有TimeStamp属性。ClientUpdateTime和LastReceivedAckRealTime都用Client本地时间赋值，和服务器时间没有关系，只表示一个绝对时间的数值。

但CurrentTimeStamp和TimeStamp表示时间戳，用于标记一个移动发生的时间，或者可以认为就是标记一个move，它的值需要在Autonomous客户端、服务器、Simulate客户端之间传输，并不是简单的绝对时间。当Autonomous客户端执行了时长为deltatime的move，CurrentTimeStamp会增加deltatime，下一个tick产生的move又会把新的CurrentTimeStamp作为自己的TimeStamp。把move相关信息发往服务器时，参数就会包括TimeStamp。之后如果服务器拒绝某次move，就会把这个move时间戳发往客户端，客户端再去SavedMoves数组里找其余move，进行重放。

如下图所示，客户端发送了两个move，时间戳为move本身属性，与服务器时间、网络传输时间等都没关系。



由于对时间戳有精度要求，而float随着数值增大，精度会逐渐降低，因此需要定期重置一下TimeStamp，由MinTimeBetweenTimeStampResets数值控制，默认240秒。这个值是要求大于两倍客户端timeout时间的，这样服务器在收到一个timestamp突然很小的rpc时，就能判定它的客户端已经重置了时间戳，而不是收到了一个很早产生的rpc。因此在修改timeout时间时这个值记得要一起修改。

**延迟发送Move**

一个Move可以被延迟一会儿，与后面的Move合并后再发往服务器，减少带宽。因此新建的Move被发往服务器前会先判断是否可以延迟发送。

首先会判断当前是否开启了NetEnableMoveCombining，以及当前Move是否能被延迟发送，会检查该Move前后MovementMode是否改变。简单的理解就是这次move是否有显著改变，比如玩家长时间向着一个方向匀速移动，那么中间的move信息其实不需要全部发往服务器，服务器可以把之前收到move中的速度作为之后的速度计算，结果应该是一样的。

然后会计算当前预期的移动更新时间间隔，有一个可配置基准值ClientNetSendMoveDeltaTime，同时根据当前网速、玩家数量、玩家是否静止等信息在基准值上做调整，得到最终间隔。如果两次Tick间隔小于更新间隔，就会延迟发送这个Move，把它存储到PendingMove属性中，留着以后处理。

**CallServerMove**

函数接受两个参数，一个是刚创建的Move，另一个是之前获取的ImportantMove，ImportantMove可能为空。不需要把Move整个都发往服务器，只需要位置、旋转、加速度等关键信息，而且这些信息会经过压缩。

首先，UE把旋转中的Yaw和Pitch压缩到一个uint32中，把Roll压缩到一个uint8中，把原本的12字节压缩到了5字节，大部分情况下玩家也察觉不到这种程度的精度损失，并且Yaw和Pitch还用的uint16压缩，有谁能分辨出0.005度与0.01度之间的差异呢？

之后，会调用ServerMoveOld，把ImportantMove发送到服务器，但只会发送时间戳、加速度和CompressFlags信息。可以先简单理解为一种冗余保险措施。

然后，如果存在PendingMove，说明两个Move无法合并，需要调用ServerMoveDual函数一次发送两个连续的Move。如果不存在PendingMove，说明发送间隔较大或已经合并了PendingMove，就调用ServerMove发送这个Move。

**ServerMoveOld**

ServerMoveOld是一种冗余措施，防止服务器新收到一个移动数据时，因为网络丢包而落后太多，导致移动判断不通过，进而纠正客户端位置。ServerMoveOld可以让服务器使用传递的加速度，粗略的从旧位置快速移动到新位置，不校验移动结果，总之就是让服务器尽快赶上。如果网络状况良好，这个函数很少会被调用到。

以上是Autonomous客户端本地移动并发送移动数据给服务器的大概流程。

还有一些细节值得注意：

**两个FSavedMove\_Character合并**

当我们在TickComponent中新建完FSavedMove\_Character，PerformMovement之前，会检查能否和PendingMove合并。

UE4没有约定哪些情况能合并，但用一些量化指标约定了哪些情况不能合并。

一些不能合并的条件：

1. 任一个move的bForceNoCombine为true
2. 包括rootmotion的move
3. 加速度从大于0变为0
4. 两次移动DeltaTime总和大于MaxMoveDeltaTime
5. 两个加速度点积超过AccelDotThresholdCombine阈值
6. 两个move的StartVelocity，一个为0，一个不为0
7. 两个move的MaxSpeed差值大于MaxSpeedThresholdCombine
8. 两个move的MaxSpeed一个为0，一个不为0
9. 两个move的JumpKeyHoldTime，一个为0，一个不为0
10. 两个move的bWasJumping状态、JumpCurrentCount、JumpMaxCount不一致
11. 两个move的JumpForceTimeRemaining一个为0，一个不为0
12. 比较两个move的CompressedFlags，包括了跳跃状态和下蹲状态，当然可以加自定义状态
13. 两个move站立的可移动表面不同
14. 两个move的开始MovementMode不同，或者结束MovementMode不同
15. 两个move的开始胶囊体半径不同，或者高度不同，一个例子是下蹲会改变胶囊体
16. 两个move的attach parent不同，或者attach socket不同
17. attach的相对位置改变了
18. 两个move的overlap数量改变

如果这些基本条件都不满足，可以初步认为可以合并。但还有一种特殊条件需要判断，就是Pendingmove的回滚位置有碰撞，这也不能合并。

合并首先需要把CharacterMovement的这一帧开始状态设置成PendingMove的开始状态，这会把UpdatedComponent的位置设成PendingMove的开始位置，包括Velocity、CurrentFloor，跳跃信息等也会设置为PendingMove状态，然后时间间隔DeltaTime会被设置为两个move之和。

之后就是修改这帧的move了，因为之前已经修改了Character和CharacterMovement的属性，因此再次调用SetInitialPosition函数，用Character重新初始化move即可。

至此，PendingMove就可以从SavedMoves数组中移除了。

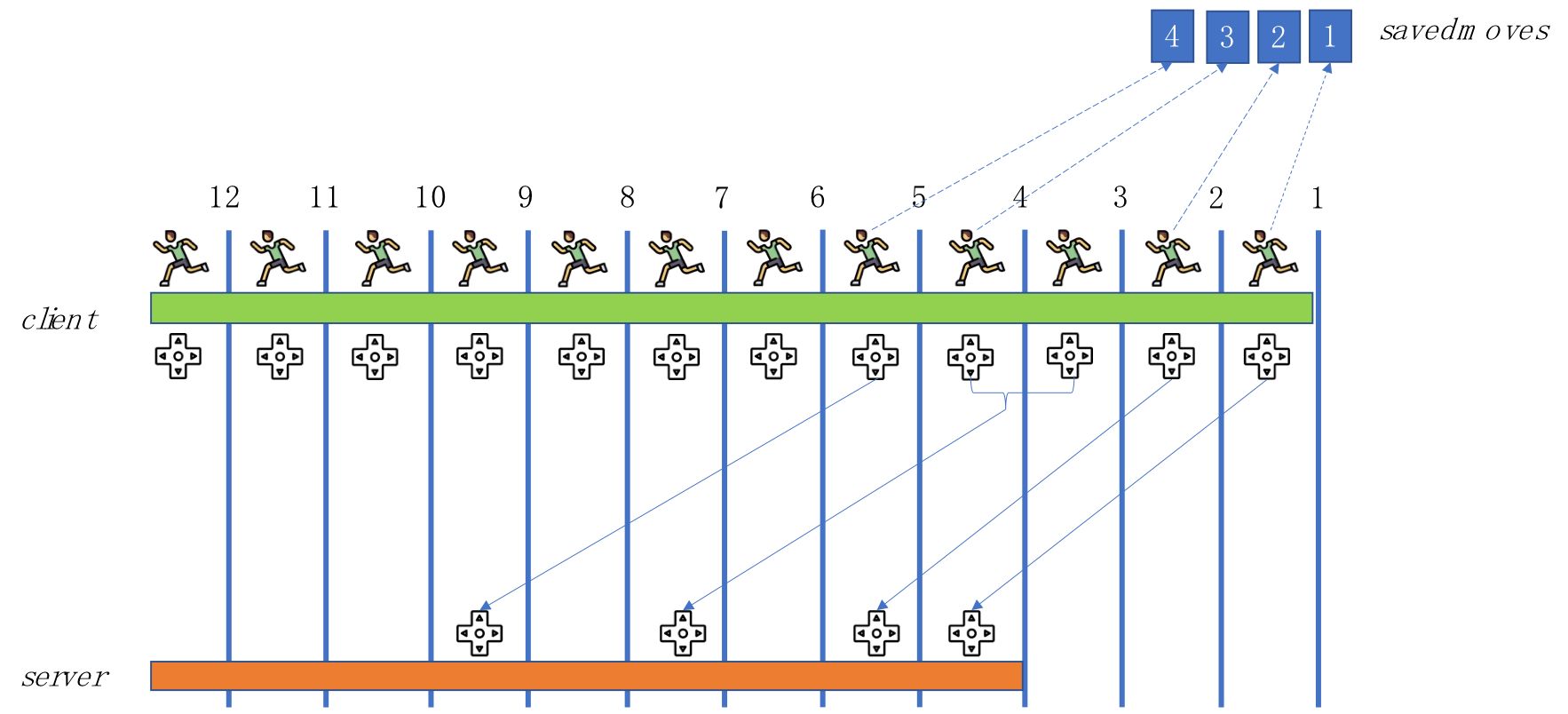
**FSavedMove\_Character缓存池**

FSavedMove\_Character数据结构本身也不小，频繁创建和销毁都不太好，因此UE使用FreeMoves数组作为缓存池。缓存池初始长度由MaxFreeMoveCount属性控制，默认96，使用过程中如果耗尽也会立即新建补充。因此获取一个新的FSavedMove\_Character需要调用CreateSavedMove接口，销毁则调用FreeMove接口，不要直接使用new和delete。

另外，SavedMoves数组也有长度限制，由MaxSavedMoveCount控制，默认也为96，如果长度到达这个阈值，就说明玩家网络情况很差，会直接把SavedMoves清空。这会对移动被服务器拒绝后的客户端重放有一定影响。

总结图例：

可以看到，客户端第三和第四帧产生的move被合并成了一个。



**Server执行移动**

本地客户端执行移动后，并不是简单的把移动结果通知给服务器就好了，而是需要在服务器上也用相同起点、速度、跳跃等信息同样执行一次移动，看和客户端移动结果是否吻合。网络游戏中所有客户端都是“不可信”的，它们可能开挂，因此服务器上的移动操作是必须的。

观察ACharacter，可以发现里面有多种ServerMove相关RPC，Client会根据本地情况发送对应RPC。

ServerMove：执行一次移动

ServerMoveNoBase：与ServerMove类似，但是不传输ClientMovementBase和ClientBaseBoneName，节省带宽

ServerMoveDual：执行两次移动，当PendingMove无法合并时，会一次把PendingMove和新的Move都发往服务器

ServerMoveDualNoBase：与ServerMoveDual类似，无ClientMovementBase和ClientBaseBoneName信息

ServerMoveDualHybridRootMotion：第一次move无root motion，第二次move有root motion

ServerMoveOld：执行一个重要的 old move

这些RPC都会以高频率发送，而且部分移动信息丢失可以接受，因此都标记为unreliable，用于节省带宽和性能。这也给移动同步带来一个挑战，就是如何在rpc不可靠情况下，通过发送一些冗余数据，让服务器移动状态尽量与客户端一致。

除了ServerMoveOld，其他rpc最终都由ServerMove\_Implementation处理，这也是服务器处理玩家移动的主要流程。

**ServerMove\_Implementation**

函数参数如下：

TimeStamp：这次移动发生的时间戳

InAccel：移动加速度

ClientLoc：移动后客户端算出的位置

CompressedMoveFlags：跳跃、下蹲等标记

ClientRoll：客户端移动后的ControlRoll，描述Controller，而不是Character

View：客户端移动后的ControlYaw和ControlPitch

ClientMovementBase:角色所站立的表面

ClientBaseBoneName：角色所站立表面的骨骼名称，如果站在skeletal mesh上

ClientMovementMode：客户端移动后的MovementMode

服务器所要做的，就是根据TimeStamp、移动加速度和moveflags，在服务端用相同方式进行角色移动，然后把结果和其他参数进行比对，判断移动是否合法。

**校验TimeStamp**

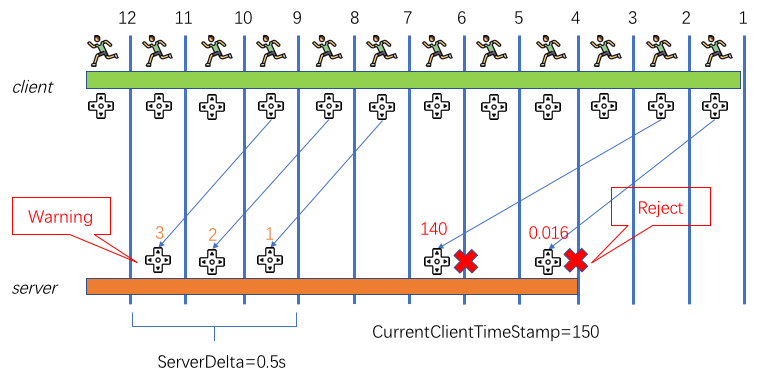
首先Server会做时间戳参数的校验。Server上对每个角色也存储了一些时间戳，在FNetworkPredictionData\_Server\_Character数据结构中有CurrentClientTimeStamp属性，表示最近处理的移动时间戳，无论移动校验是否通过，都会被更新，也会定期重置。

先把TimeStamp和CurrentClientTimeStamp做比较，初步校验以下情况：

1. TimeStamp小于CurrentClientTimeStamp
2. 差异大于MinTimeBetweenTimeStampResets\*0.5， 即大于timeout，且排除重置情况，说明收到的是很早之前的rpc
3. 两者差异过小，小于MIN\_TICK\_TIME，不然会有些异常情况

这些校验通过了，只能说明TimeStamp没有明显异常，还需要做进一步的绝对时间校验，处理客户端用加速软件等情况，毕竟不能完全以客户端时间为准。打开bMovementTimeDiscrepancyDetection设置才会检查，默认关闭。服务器会记录一份收到rpc的服务器绝对时间，如果来自客户端的时间戳间隔与服务器绝对时间间隔差异过大，就会认为这个客户端有问题，可以选择记下log，并且强制这个客户端更新到服务器位置。

如下图所示，当服务器收到比当前时间戳小的move，会直接忽视，当收到的时间戳增速比服务器时间增速快很多时，会进行报警，可选择拒绝等处理。



**执行移动**

时间戳校验通过后，就可以进行服务器的移动了。首先从参数中还原数据，因为加速度、ControlRotation信息被压缩传输，要把它们还原为FRoataor和FVector，还有蹲下、跳跃信息，也要从CompressedFlags里解压出来。对于ControlRotation，会不做校验，直接应用于Controller。之后会执行PerformMovement方法，在服务器上也做一次移动。

**移动后校验**

移动完成后，需要把服务器上移动结果和客户端发来的结果做比对，判断要接受这次移动还是拒绝，并通知客户端，由ServerMoveHandleClientError函数完成。

首先，通知客户端移动判定结果，也需要向客户端发送rpc，因此服务器会参考时间间隔和客户端带宽情况，避免频繁向客户端发送判定结果rpc，无论ack还是拒绝。

间隔检查通过后，就要执行位置检查，检查包括两部分：MovementMode和location。如果服务器移动后的MovementMode和客户端不同，检查肯定不通过。之后检查location，如果与客户端location差异太大，则为校验不通过，需要客户端调整位置；如果位置差异极小，也会选择容忍这个差异，算校验通过。

**位置校验通过**

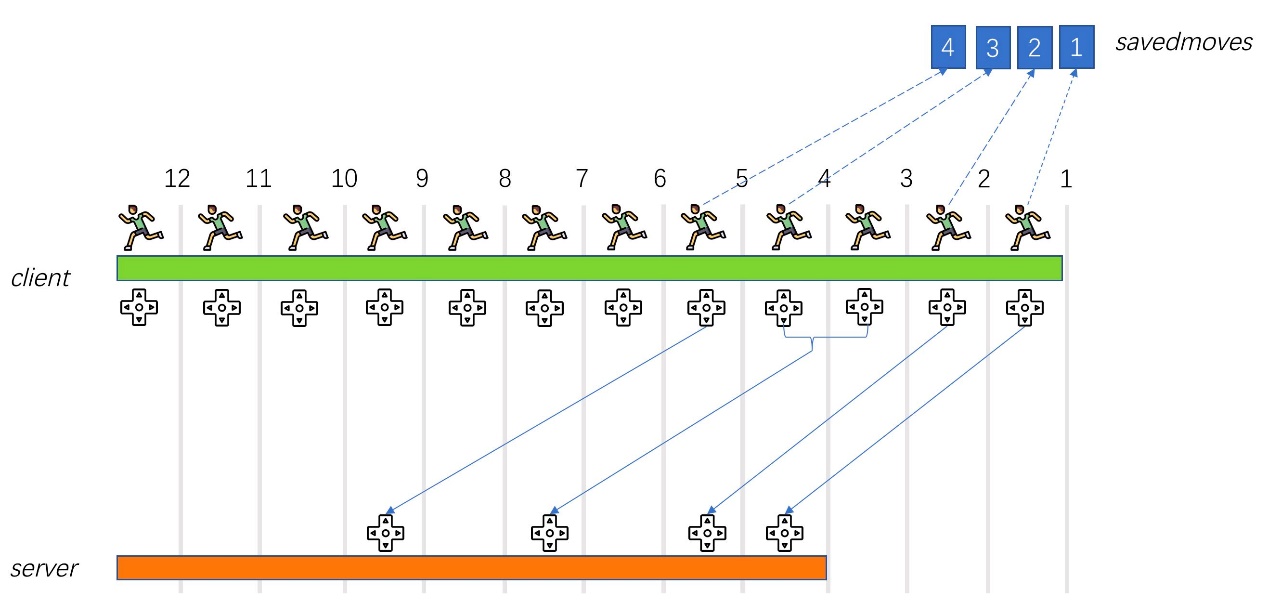
如果位置校验通过了，可以选择是否直接使用客户端位置，因为服务器位置和客户端位置相差并不大。如果服务器不用客户端位置，虽然这次检查通过了，但下次的位置误差可能会累加，导致调整。服务器使用客户端位置，会减小之后的误差，但存在作弊的可能性。引擎默认不使用客户端位置。

校验通过后，需要使用ClientAckGoodMove rpc通知客户端。当然，这个rpc也被标记为unreliable，即使一个因为网络丢包漏掉了，只要后面有被客户端收到的即可。这个rpc会频繁触发，因此可以设置最小发送间隔，默认为0.1s。

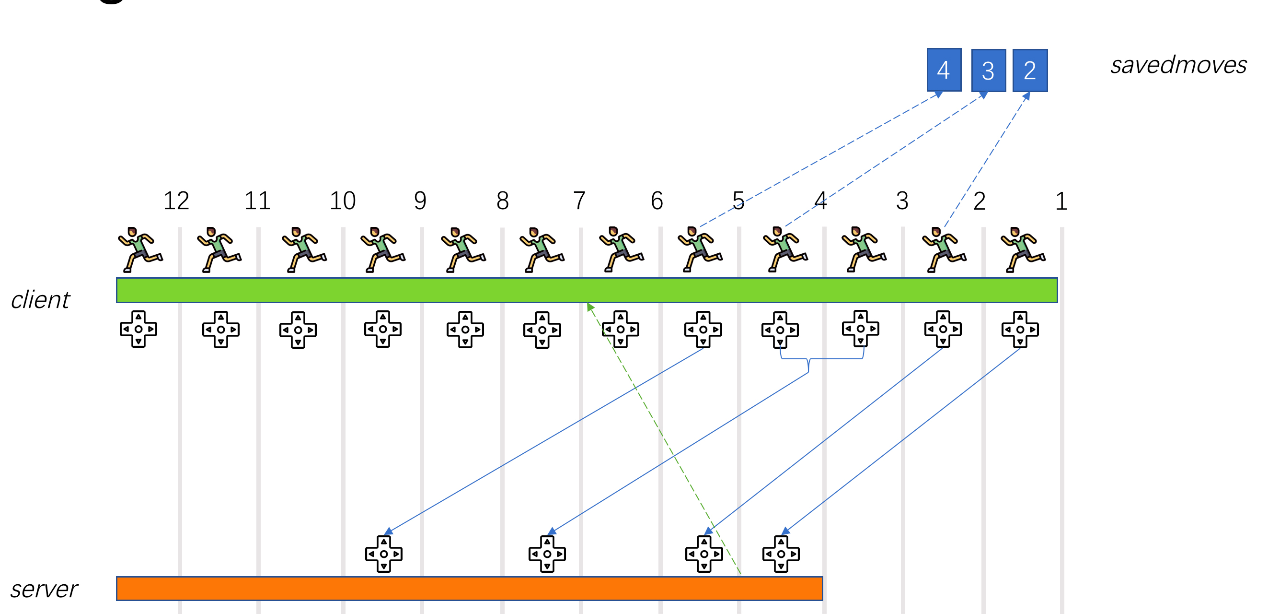
ClientAckMove的参数只有一个，就是PendingAdjustment的TimeStamp。Autonomous客户端收到rpc后，根据TimeStamp从SavedMoves数组里找到对应的move，把它作为当前的LastAckedMove，然后把SavedMoves中TimeStamp之前的Move都删除，表示之前的move都被ack了。

示例：

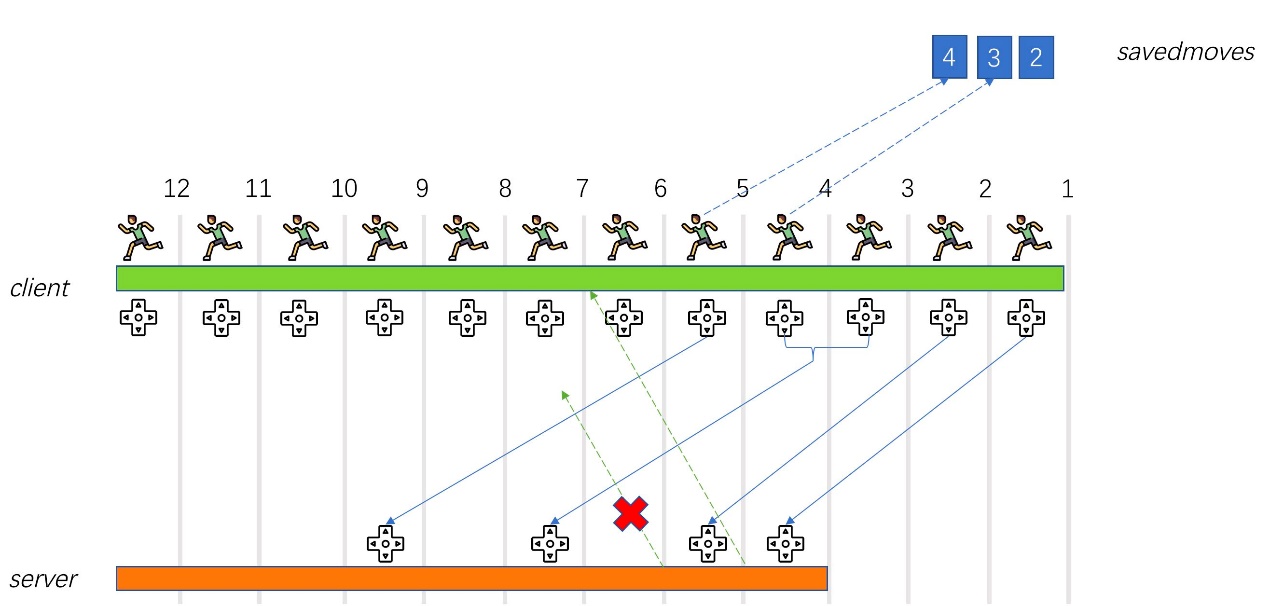
首先客户端发送了四个move给服务器



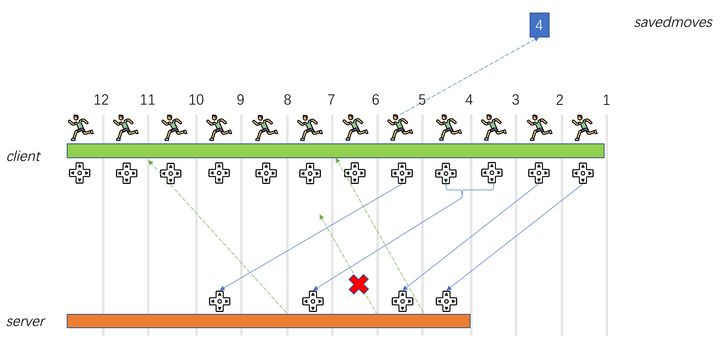
服务器针对move1发送ack给客户端，客户端收到，从savedmoves里移除move1



服务器针对move2发送ack给客户端，但由于网络丢包，客户端没收到



服务针对move3发送ack给客户端，客户端收到，发现时间戳对应move3，说明move2也校验通过，因此从savedmoves中移除move2和move3



**位置校验不通过**

位置校验不通过，就要通知Autonomous客户端更正自己的位置。FNetworkPredictionData\_Server\_Character数据结构有PendingAdjustment属性，用于存储调整客户端位置所用信息，主要属性如下：

TimeStamp：move的时间戳

DeltaTime：move用时

NewLoc：位置

NewVal：速度

NewRot：UpdatedComponent的朝向，不是controlrotation

bAckGoodMove：是否是goodmove

校验不通过，会调用ClientAdjustPosition方法通知客户端调整位置，和ClientAckMove一样，都在SendClientAdjustment函数中处理。

如果移动后速度为0，会使用ClientVeryShortAdjustPosition版本rpc，如果不为0则使用ClientAdjustPosition版本rpc。前者只是在rpc参数里节省了速度值而已，其实和ClientAdjustPosition没有区别。

ClientAdjustPosition函数

首先，rpc的参数就是PendingAdjustment中属性。之后客户端会把位置、速度、MovementMode都设置成和服务器相同，然后把SavedMoves中TimeStamp对应move及之前move都ack，并删除，它们已经没用了。最后设置标记位bUpdatePosition，表明自己被服务器纠正了位置。

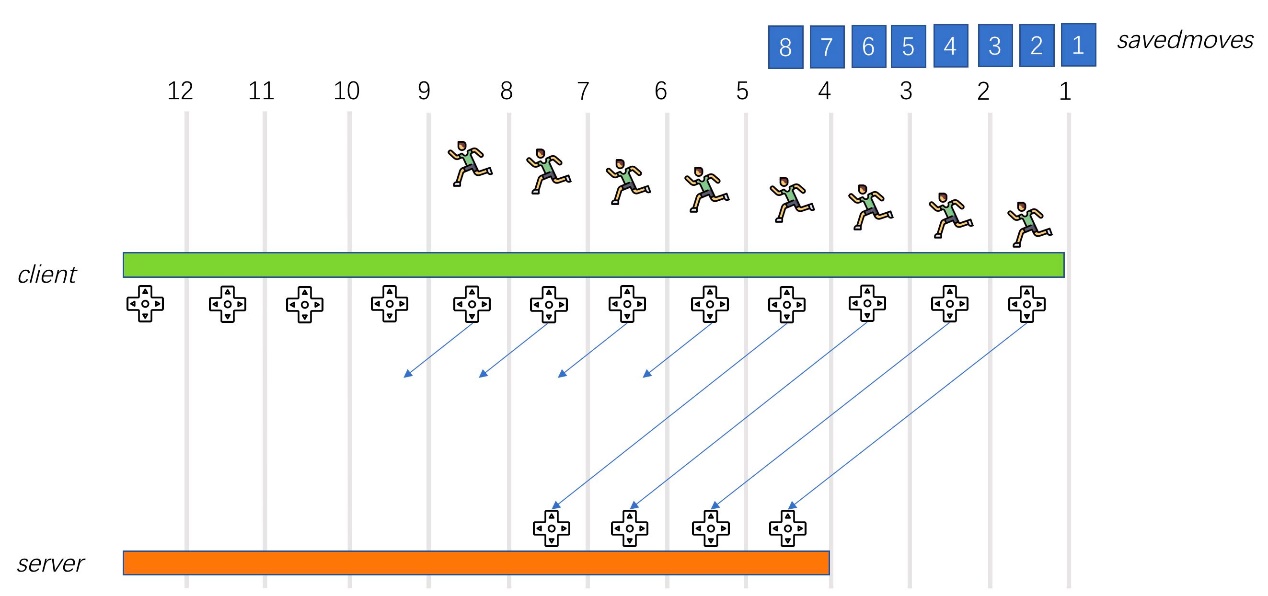
bUpdatePosition属性

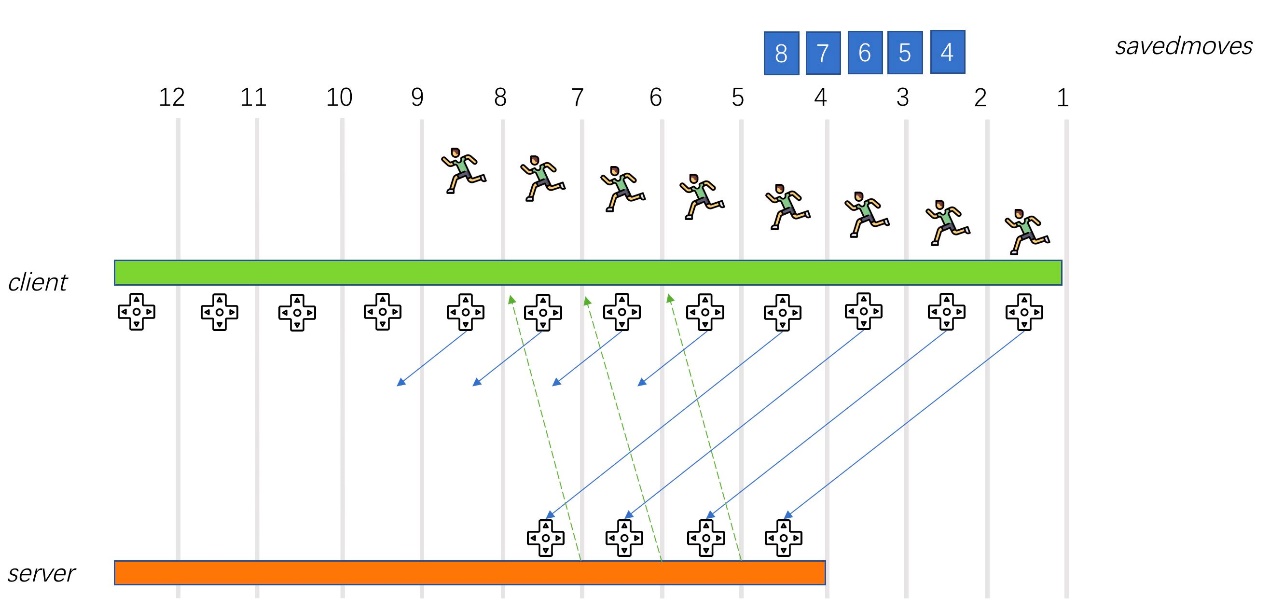
在Autonomous客户端执行TickComponent时，会检查bUpdatePosition是否为true。如果是，就要重播当前savedmoves中的所有move，做法也比较简单，先提取move信息，然后使用MoveAutonomous移动即可，这些移动不需要再发送servermove rpc了。重播结束后，玩家已经在被纠正位置的基础上把后续输入都重演了一遍，后续位置可能会和服务器算的保持一致。接着客户端继续接受输入并正常移动，可能还会收到服务器纠正位置消息，因为这些move在重播前已经发了server move rpc，这些rpc里location也是错的，服务器会继续纠正，但角色后续客户端位置与服务器位置误差会逐渐变小，并最终保持一致。

为什么要重播后续move呢，仅把角色拉回可以吗？因为服务器纠正的是一个RTT前的位置，如果只是把角色拉回，从数据正确性上来说没问题，但玩家会感觉自己突然回到来RTT前，而且中间输入操作丢失了，会不够流畅。因此重播是一种比较好的实践做法。

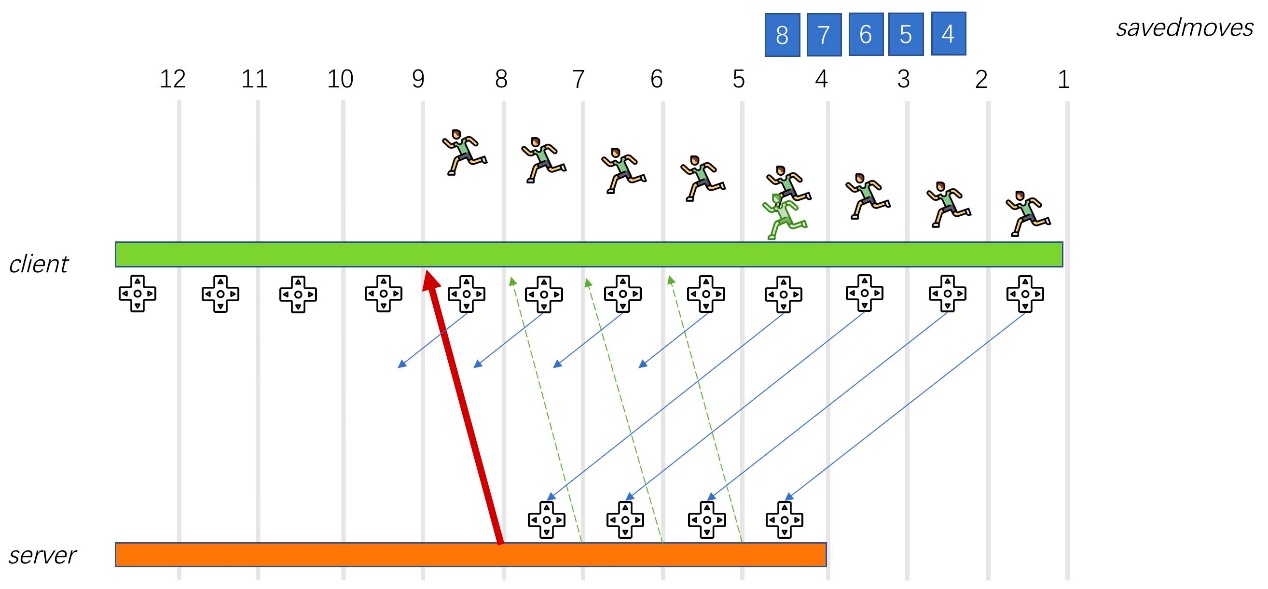
示例：

前3个move校验通过

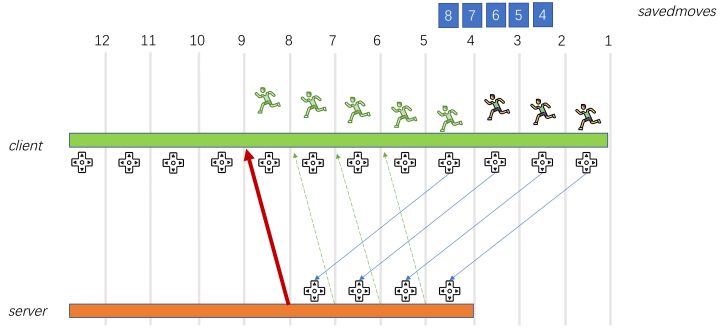




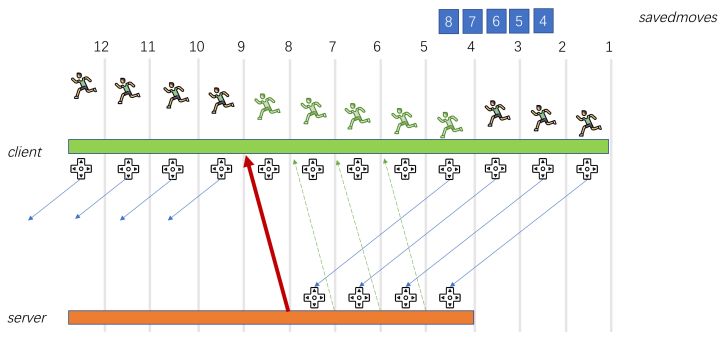
第四个位置校验不通过，服务器判定应该处于绿色位置，可能是撞墙了



客户端把玩家位置调整为服务器通知位置，然后重放savedmoves之后的移动，重放的移动不会发送通知给服务器



之后客户端正常接受玩家新的输入并移动



这就是服务器纠正位置的大体流程。

**Server同步移动给客户端**

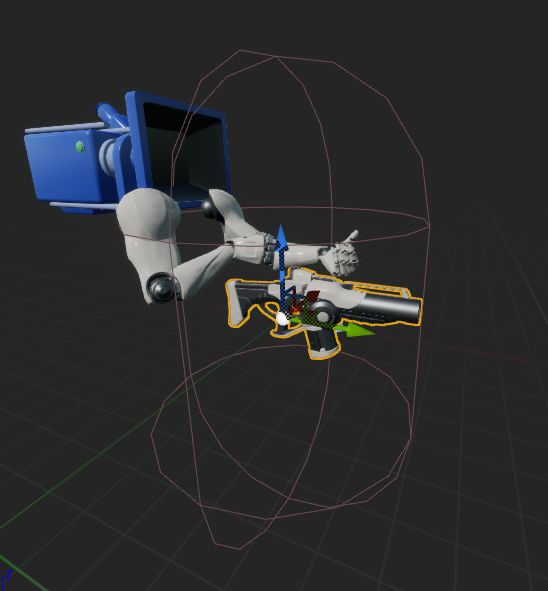
**Actor基本同步方案**

Actor自身就支持移动同步，打开ReplicateMovement开关后，当Actor的RootComponent位置、转向等数据发生变化时，就会把数据同步给Simulate客户端。同步的数据结构为FRepMovement，当Simulate Actor收到同步数据后，会简单的设置自己的位置和朝向。这么做可以实现简单的同步，但Simulate Actor的移动间隔取决于移动数据同步间隔，移动数据显然不会每帧都同步下来，而且间隔也不是固定的，因此当Actor执行一段连续的移动，Simulate Actor位置会发生间歇性闪现，人形玩家这么移动就显得很糟糕了。

**Character移动同步**

针对Actor基本同步模式的不足，CharacterMovementComponent针对性的做了表现平滑处理，让Simulate角色移动在与服务器一致的情况下，尽量显得平滑自然。

Character主要有两个组件，Capsule和mesh，Capsule是Character的RootComponent，用于处理碰撞，它的位置代表了Character当前的位置，但它是看不见的。mesh用于角色模型的显示，玩家能看到的是mesh，因此移动平滑主要针对mesh进行处理。



**Simulate Character收到移动同步**

Character收到移动同步后，主要在SmoothCorrection函数里处理相关逻辑，同一帧中，收到移动同步数据是在ComponentTick之前发生的。虽然函数名叫SmoothCorrection，但真正做的事情却不简单。

首先，服务器同步下来Actor位置与本地Actor位置极大概率是不同的，把它们间距离记为Dist。然后Sinulate客户端维护了一个MeshTranslationOffset向量，表示当前Sinmulate上mesh位置与服务器同步Capsule位置的相对差异，会把Dist累加上去，之后平滑处理的目的就是让这个值逐渐变小。如果Dist太大，超过了平滑失效距离，默认20cm，就把MeshTranslationOffset归零，表示这次不做平滑了，mesh直接设置到新的位置，尽快与服务器位置同步。

**收到数据后Location&rotation更新**

接下来就要更新Simulate Capsule了，分Linear平滑和Expontial平滑两种情况。

对于Linear平滑，会记录下当前本地rotation和服务器新同步的rotation，之后会在它们之间做插值处理。然后把Capsule位置更新到新同步位置，mesh不动。

对于Expontial平滑，会记录下rotation的变化差异，然后把Capsule位置和rotation都更新到新同步的，mesh也不动。

**TimeStamp处理**

时间戳是移动同步中永远避不开的话题，但移动同步数据结构FRepMovement只会把角色当前的位置信息同步给客户端，时间戳通过单独的ReplicatedServerLastTransformUpdateTimeStamp变量同步，这个TimeStamp只是和同步的move时间戳相近，并不完全对应。

Simulate客户端维护了两个TimStamp，这些时间戳仅针对Linear插值平滑起作用。

SmoothingClientTimeStamp：Simulate本地当前平滑移动时间戳

SmoothingServerTimeStamp：当前servermove的时间戳

在此不做过多展开，知道它们即可。

**插值平滑**

需要平滑的有Capsule的Rotation、mesh的location和rotation，这些都是视觉上能观察到的。每帧TickComponent时，会执行SmoothClientPosition，平滑都在里面处理。

首先，有两种平滑模式可选，Linear和Exponential。

**Linear插值平滑**

**关键词：匀速运动，周期准确，受网络波动影响**

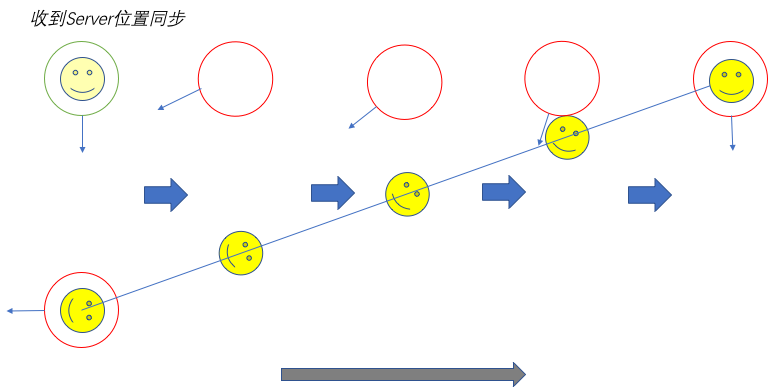
Linear使用线性插值进行平滑，视觉上位移和转向变化为匀速运动。

**插值周期**

插值周期在之前已介绍过，主要通过服务器同步下来的时间戳ReplicatedServerLastTransformUpdateTimeStamp进行计算。不使用收到两次移动同步的间隔，是因为这个间隔包括了数据包在网络上传输的时间，网络传输时间是不固定的，这样可以避免网络传输时间的干扰。ReplicatedServerLastTransformUpdateTimeStamp也有两种计算方式，一种为使用客户端发送move的时间戳累加，另一种为直接使用服务器当前时间，可通过控制台参数设置，理论上第一种更准确，引擎默认也使用第一种。

首先更新代表当前移动进度的SmoothingClientTimeStamp，加上这次tick的DeltaSeconds，最大不会超过LastCorrectionDelta的1.15倍，防止客户端走的太前。然后计算插值比率，值为SmoothingClientTimeStamp/LastCorrectionDelta，结果在[0,1.15]之间。因为Linear模式下，收到移动同步数据时只更新了Capsule的location，因此接下来需要根据插值比率，同时对Capsule和mesh做插值平滑。

根据插值比率，算出新的MeshTranslationOffset和MeshRotationOffset，作为新的location和rotation偏移值。之后，对Capsule设置新的rotation，此时mesh的rotation会跟着一起改变，然后对mesh设置新的相对Capsule位置。



至此，Linear平滑操作结束。

Linear插值是在两个位置之间线性插值，呈匀速运动，插值周期为服务器同步的移动时间戳，因此插值周期比较准确。如果服务器同步周期出现波动，角色插值平滑会出现一些问题。比如下一个网络同步间隔变大，角色在插值平滑结束后会不动，rotation表现比较明显。

**Exponential插值平滑**

**关键词：计算简单，先快后慢**

也称为指数插值平滑，与Linear平滑不同，Exponential平滑使用固定插值周期，不使用服务器同步时间戳，而且移动也不是匀速的，当前位置与目标越远，插值的位移越大，整体实现更加简单。

不需要服务器同步时间戳还有一个好处，可以节省带宽。

在收到服务器同步数据后，Capsule的location和rotation被立即设置为新的值，需要插值处理的是mesh的location和rotation。

以location为例，每一次tick时，采用如下公式进行插值：

LocationOffset = LocationOffset\*(1.f - DeltaSeconds / SmoothLocationTime)

LocationOffset为mesh当前位置偏移，DeltaSeconds为tick间隔，SmoothLocationTime为插值周期。如果客户端以固定速率tick，则括号中数值不变，公式可改写为：

LocationOffset = LocationOffset\*ΔP

另LocationOffset0代表刚收到同步位置时的原始偏移，LocationOffsetN代表收到同步数据后第N此tick时的偏移，则

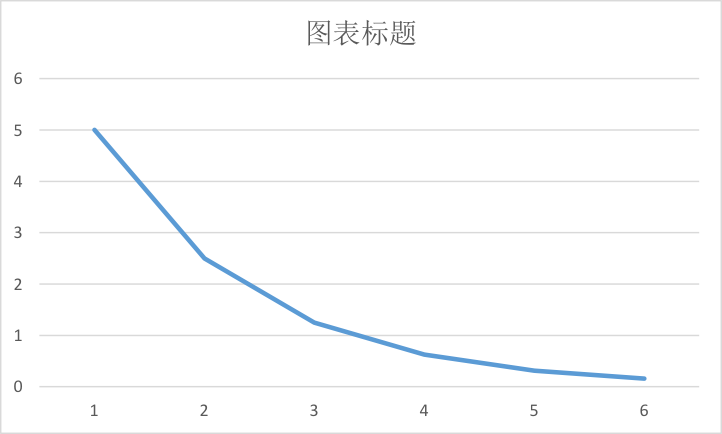
LocationOffset1 = LocationOffset0\*ΔP

LocationOffset2 = LocationOffset0\*(ΔP^2)

LocationOffsetN = LocationOffset0\*(ΔP^N)

因此称为“指数”。

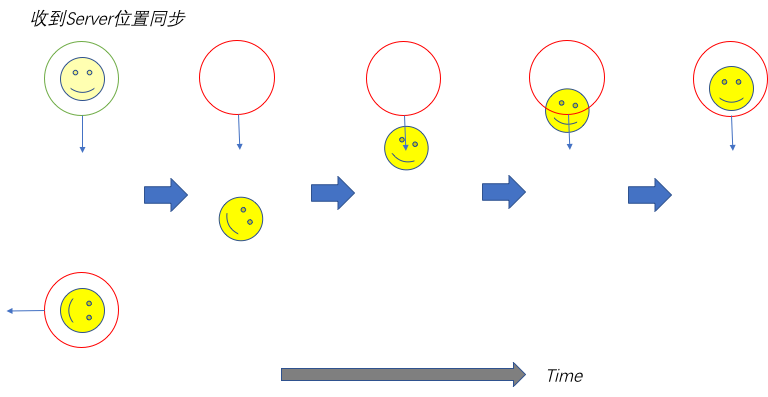
下图展示了指数插值下tick次数与位置差异的关系，可以发现是指数减小的。



Exponential插值其实没有明确的“插值周期”概念，按照插值过程，当前location永远也不会到达目标location，只会无限接近，所以当两者差异小于一定阈值时，会认为插值已经完成了。Exponential插值做的，是让当前mesh置平滑移动到目标位置，不在意Autonomous客户方在两个位置的真实时间间隔，只要mesh一直在动即可，看起来就是平滑的。

实际上，Simulate客户端也无法知道Autonomous客户端到底在两个位置之间是怎么运动的。

fps游戏中的一次视角转动通常是先快后慢的，因此使用Exponential插值可能更合适些。

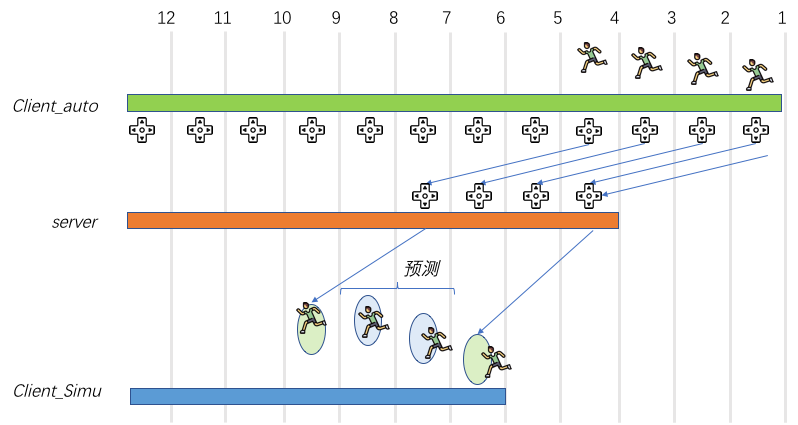


**Simulate客户端预测**

Simulate客户端收到的移动数据，相比移动刚发生时，理论上位置已经延迟了一个RTT，收到即落后。如果在此基础上进行平滑移动，平滑目标为服务器同步最新位置，那么又落后了一个平滑周期的时间。因此为了使Simulate客户端上角色更接近游戏实时状态，UE4允许Simulate角色进行本地预测移动。在每帧TickComponent时，对Simulate客户端会执行SimulateMovement，获取当前速度，大部分情况当前速度就是移动同步下来的玩家速度，然后把加速度设为和速度方向一样，大小为1，因此rotation并不会改变。移动会忽略MovementMode，且忽略摩擦力等外力对速度的影响。直到下一次收到服务器位置更新，再进行新的预测移动。

预测和插值会同时进行，预测改变的是世界transform，插值改变的是相对transform，不冲突。

加入预测后，Simulate客户端上角色移动落后就只有一个RTT了。



**一个实现细节：关于FVector\_NetQuantize**

移动同步涉及到很多实现细节，没法面面俱到的介绍，在此仅把FVector\_NetQuantize作为其中一个示例进行细致分析。

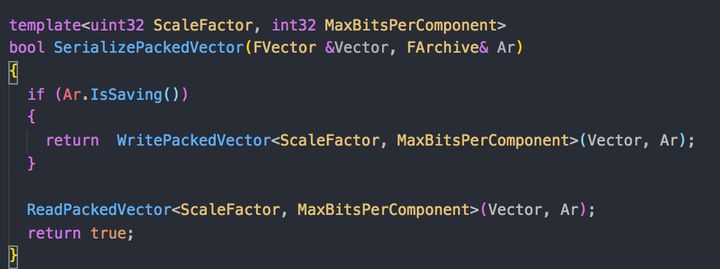
移动同步中的位置、加速度、速度都可以用FVector表示，内含3个float类型变量，总共12个Byte。相对于游戏中大部分使用场景，float的表示范围是大大溢出的，但是float遵循IEEE754标准，在网络上传输float无法使用自适应Bit流来减小数据量，导致直接传输FVector就必须用12个Byte，有些浪费。

FVector\_NetQuantize数据结构就是为了解决这个问题，它使用了有损压缩技术。在进行网络序列化时，会把向量中的float转换成int传输，减小数据量，反序列化时，再把int恢复成float，再组合成FVector，会损失一些精度，但大部分情况都可接受。

FVector\_NetQuantize数据结构有多个版本，我们常用的有FVector\_NetQuantize10和FVector\_NetQuantize100，后面的数字代表float转换成int时，精确到小数点的位数，显然后者更高。它们表示范围有限制，比如FVector\_NetQuantize10中数字表示范围是2^24/10，即+/-1677721.6，至于为什么会在接下来介绍，使用时留意此范围即可。如果FVector代表角色位置，长度单位是厘米，那么精确到1/100厘米对游戏而言是完全OK的。

**序列化**

FVector\_NetQuantize序列化会使用SerializePackedVector函数



函数接受两个模版参数，第一个是缩放大小，即10、100，第二个是转换成int后可使用的最大bit数，限制了表示范围。FVector\_NetQuantize10在序列化时，两个参数分别为10和24，因此表示范围是2^24/10。

序列化步骤如下

1. FVector乘以ScaleFactor，进行放大
2. 把float转换成int
3. 计算表示三个int绝对值+1（正数）所需最大位数，记为Bits，在MaxBitsPerComponent处截断
4. 计算偏移Bias=1<<(Bits+1)，然后把三个int都加上Bias，这是为了把负数都变成正数传输，这样才能用自适应Bit流
5. 计算上限Max=1<<(Bits+2)，并用Max-1对int数值进行截断
6. 先向数据流写入Bits，表示后续数字的最大位数，再依次写入三个int值，每个int值要求所用bit位数相同，不满的用0填充，完成序列化

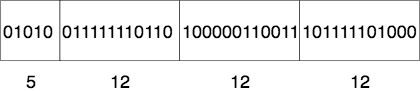
例子：使用FVector\_NetQuantize10传输(-1.0, 5.1, 100.0)

首先，把它们乘以ScaleFactor，得到(-10, 51, 1000)

然后计算表示它们需要的最大位数，为10，2^10=1024，因此Bias = 2048

之后三个int都加上Bias，得到(2038, 2099, 3048)

依次向数据流写入10和(2038, 2099, 3048)，此时数据流为：



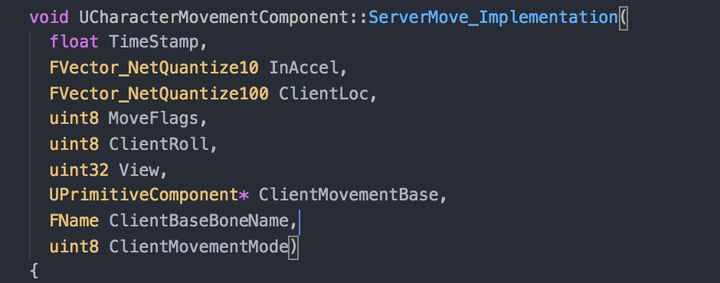
总共使用了41位，原始的三个float需要128位，仅为其32%大小。留意10和2038的第一位0填充。

**反序列化**

了解了序列化过程，反序列化也不难理解了。

首先知道这是FVector\_NetQuantize10类型数据，因此10和24这两个数字是确定的。于是可以从Bit流头部取出5位，得到“Bits“，从而得到上限“Max”和”Bias“，以及之后每个数字所用位数。剩下就是依次获取三个int值，然后减去Bias，并除以ScaleFactor进行还原。

实际案例：ServerMove接口



加速度InAccel使用FVector\_NetQuantize10存储，位置ClientLoc使用FVector\_NetQuantize100存储。

**总结**

本篇文章简要介绍了UE4移动组件如何实现移动网络同步，其中Autonomous客户端需要在立即响应玩家输入的同时，保持位置与服务器校验一致；服务器需要进行移动校验和纠正，并把移动同步给其他客户端；Simulate客户端需要使用插值和预测技术来呈现一个平滑的移动过程。移动同步的一大影响因素为网络的不确定性，移动组件的很多机制（比如ServerMoveOld()），都是为了能在不稳定的网络中实现良好的移动同步功能。除此之外，文章最后还介绍了一种实用的数据同步流量优化手段。