一、部分核心概念

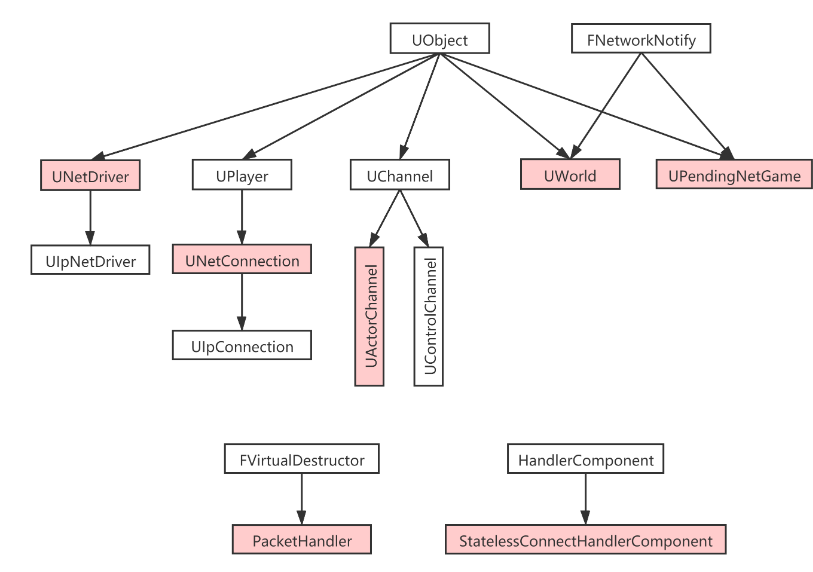
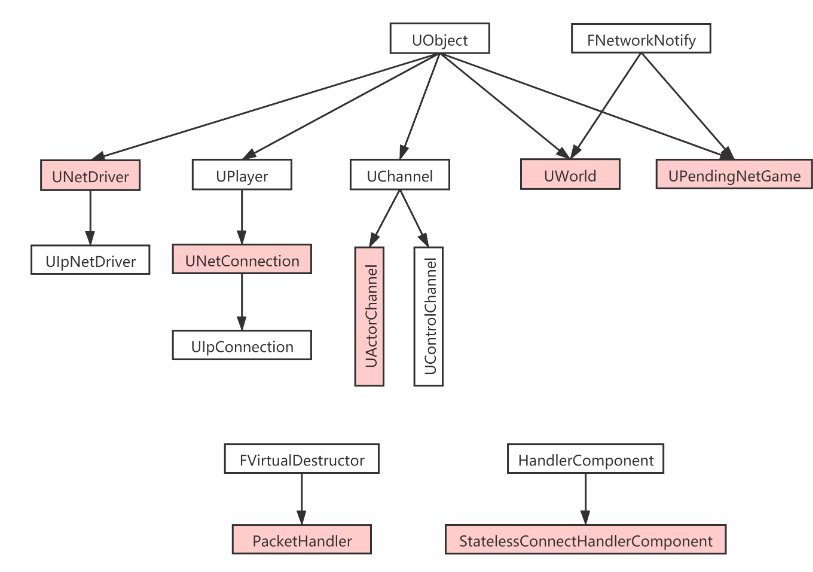
NetDriver:管理NetConnections,服务器NetDriver将维护一个NetConnections列表，每个连接代表一个连接的玩家。

NetConnection:在服务器和客户端上，NetDriver负责接受来自网络的数据包并将这些数据包传递给适当的NetConnection

Packet:是在CS网络连接之间发送的数据，由Packet包的元数据（如报头信息和确认ACK）和Bunches组成

Bunch:是在CS网络连接之间发送的数据，当一个连接收到一个数据包时，数据包将被分为单独的Bunch，这些Bunch被传递到单独的通道进一步处理，

一个Packet可以含Bunch与否， 当一个 Bunch 太大时，在传输之前会把它切成许多小 Bunch，这些 Bunch 将被标记为 PartialInitial, Partial 或 PartialFinal。利用这些信息，在接收端重新组装 Bunch



二、基本概念

NetDriver:创建Connection

Connection:关联了NetDriver、PackageMap、UChannel数组和PlayerController,一个客户端到服务器的一个连接叫一个ClientConnection,服务器到客户端的连接叫ServerConnection。

Channel: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：CS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

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

UPackageMap:生成与维护Object与NGUID的映射，负责Object序列化。每个Connection对应一个。

PacketHandler：网络包预处理：比如加解密，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，和握手组件StatelessConnectHandlerComponent

FObjectReplicator：属性同步的执行器，每个Actorchannel对应一个FObjectReplicator，对应一个对象实例。设置ActorChannel通道的时候会创建

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，所有的Properties都被LayoutCommands表征，LayoutCommands含有：1，潜在的数据类型 2,Data在内存中如何存放 3，Data如何被序列化 4，Data如何进行对比

（在Object Struct之间） 5，这个Data在改变时是否触发RepNotifies 6,这个Data是否Conditional（有些情况下不发给某些连接）

命令被分为2种类型：Parent Commands (@see FRepParentCmd) and Child Commands (@see FRepLayoutCmd).

class TestObject

{

    TArray<Arr> TestArr; // 1 Parent Command ,1 Child Command

    UStruct\* struct; //1 Parent Command

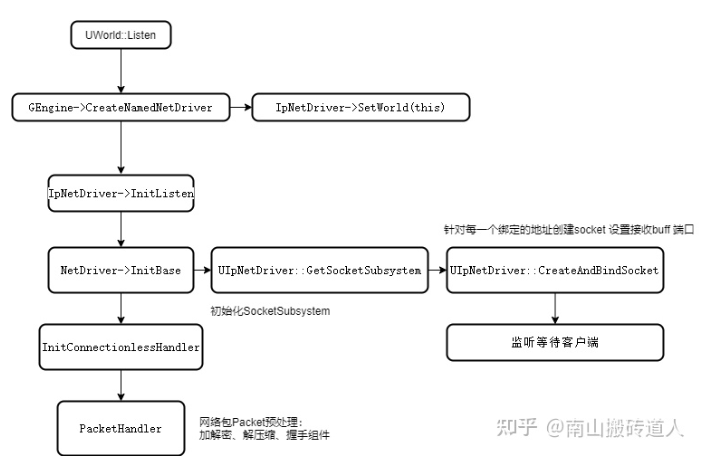
}

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。

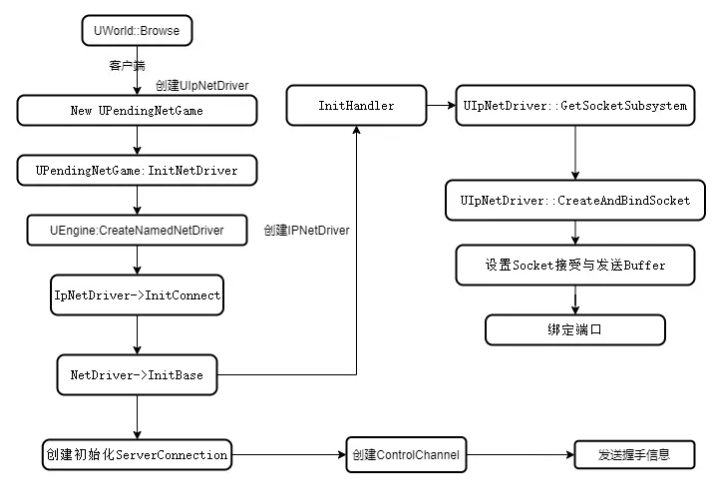
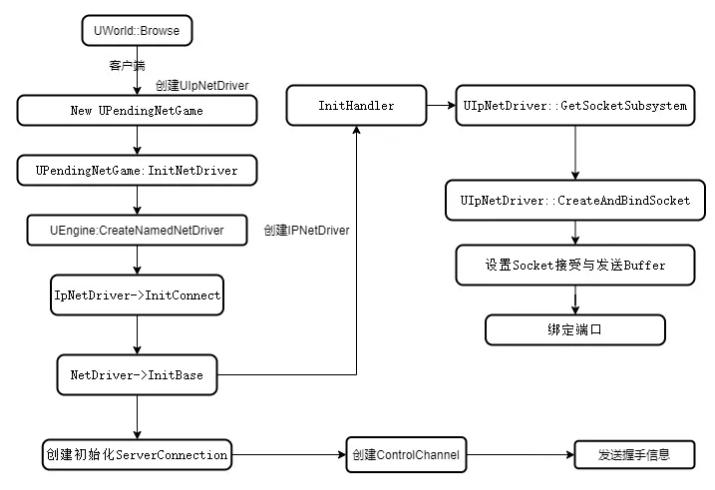
FReplicationChangelistMgr：存放当前的Object对象，保存属性的变化历史记录

三、连接流程

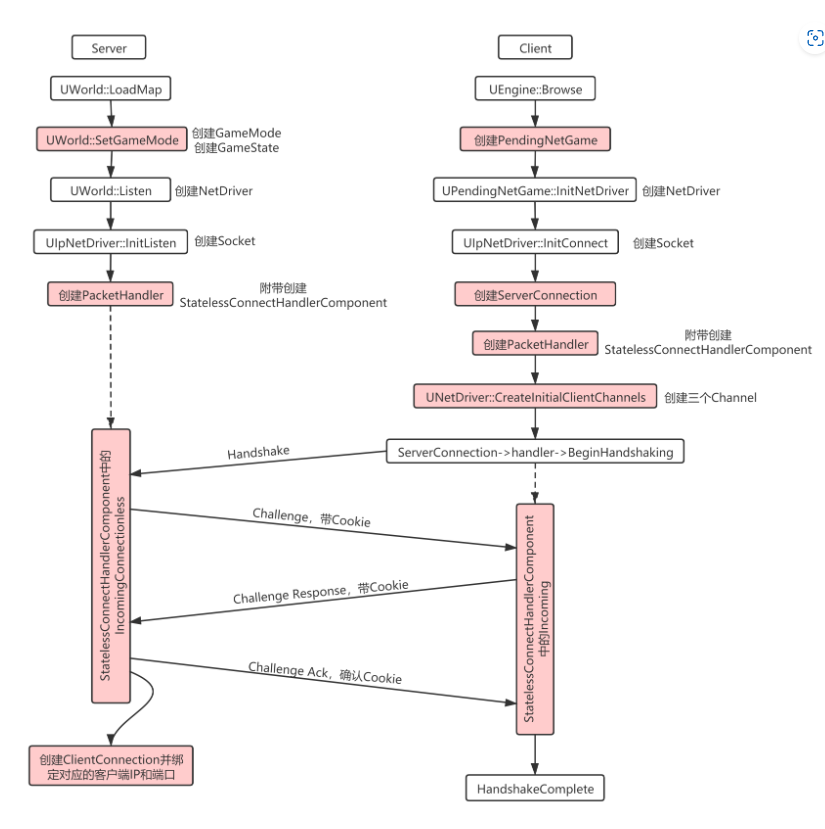
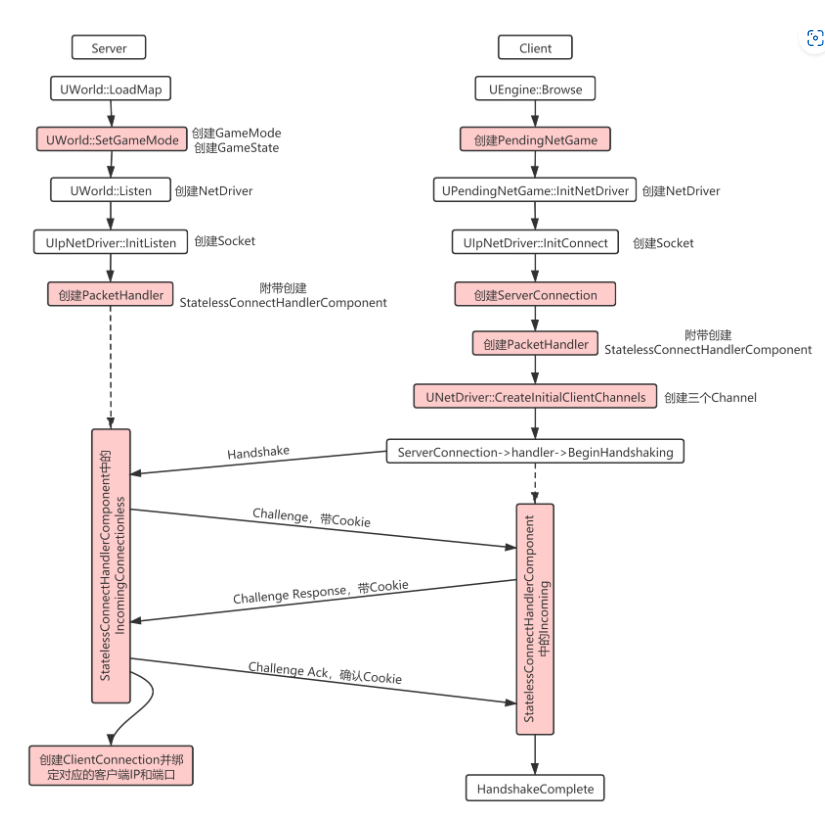
Server监听创建：



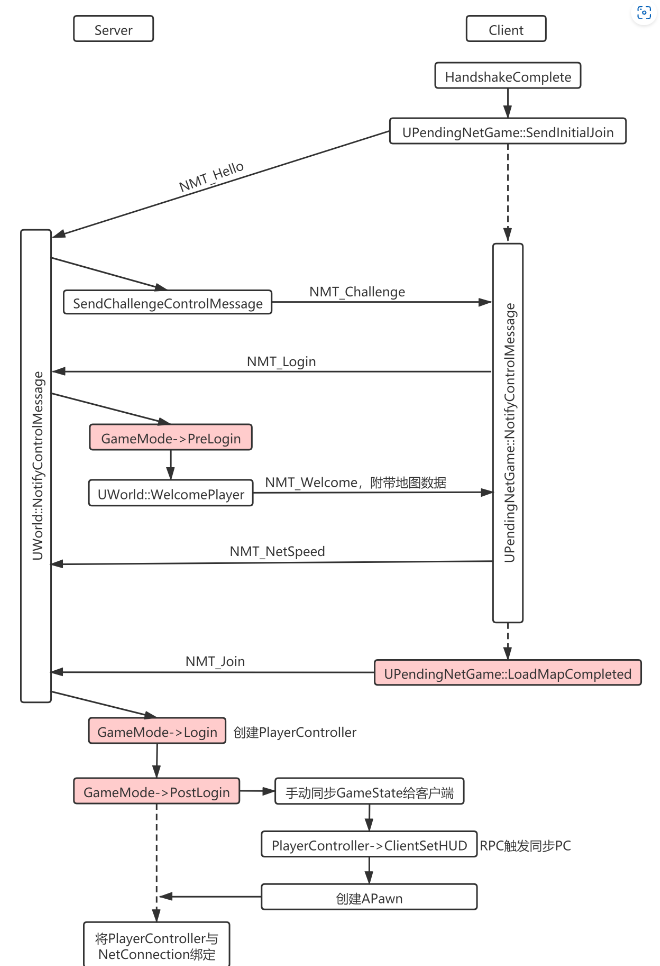
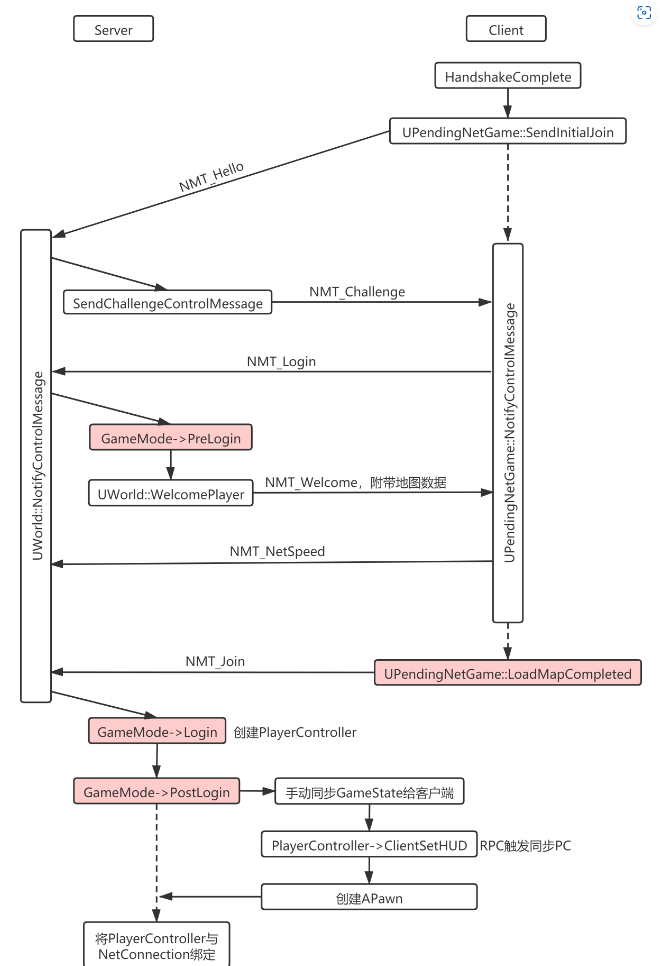
Client连接创建



Server和Client握手过程：

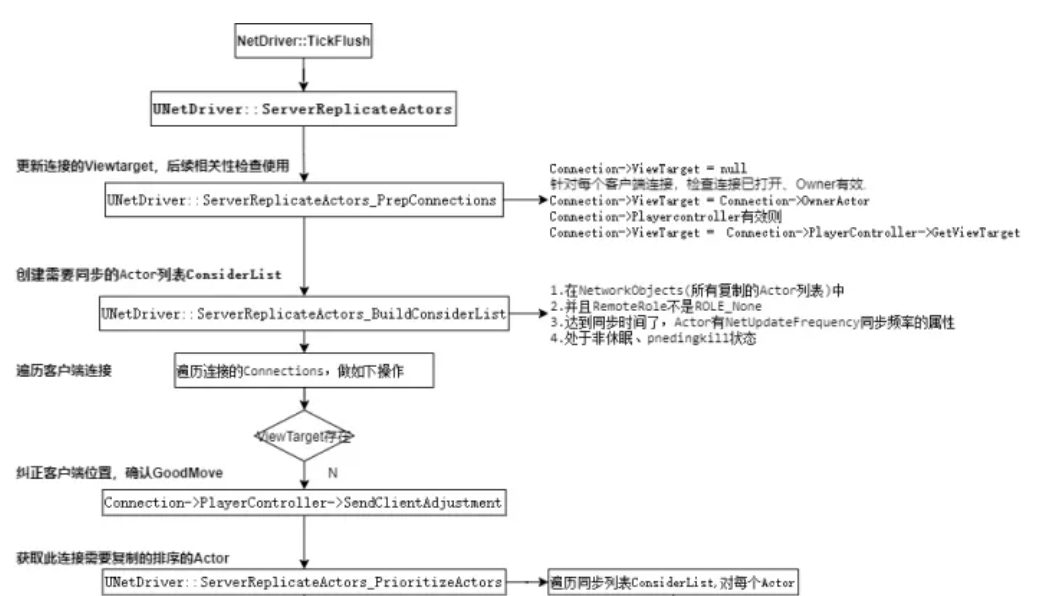
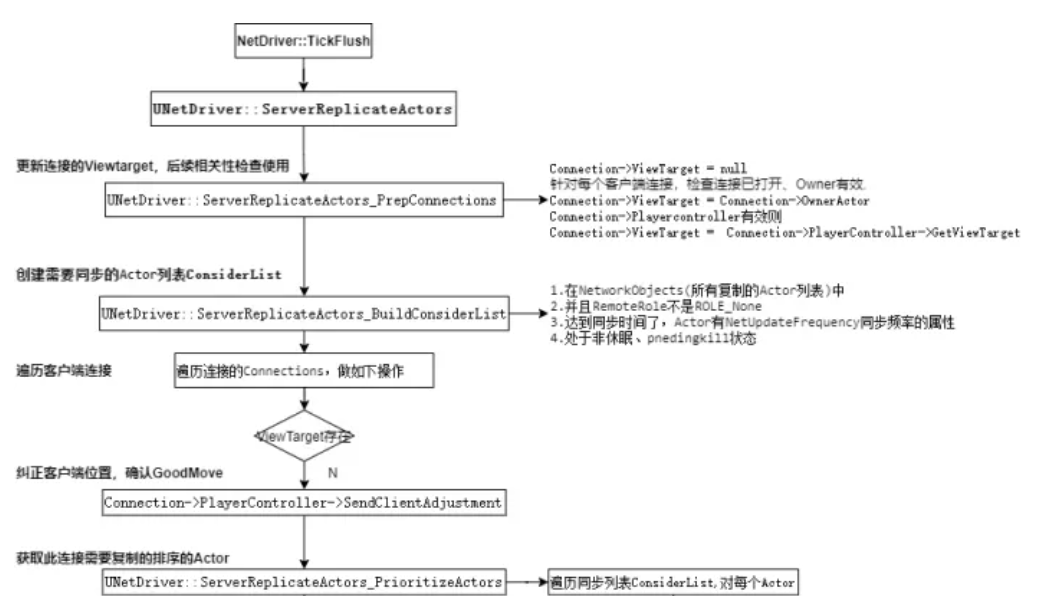


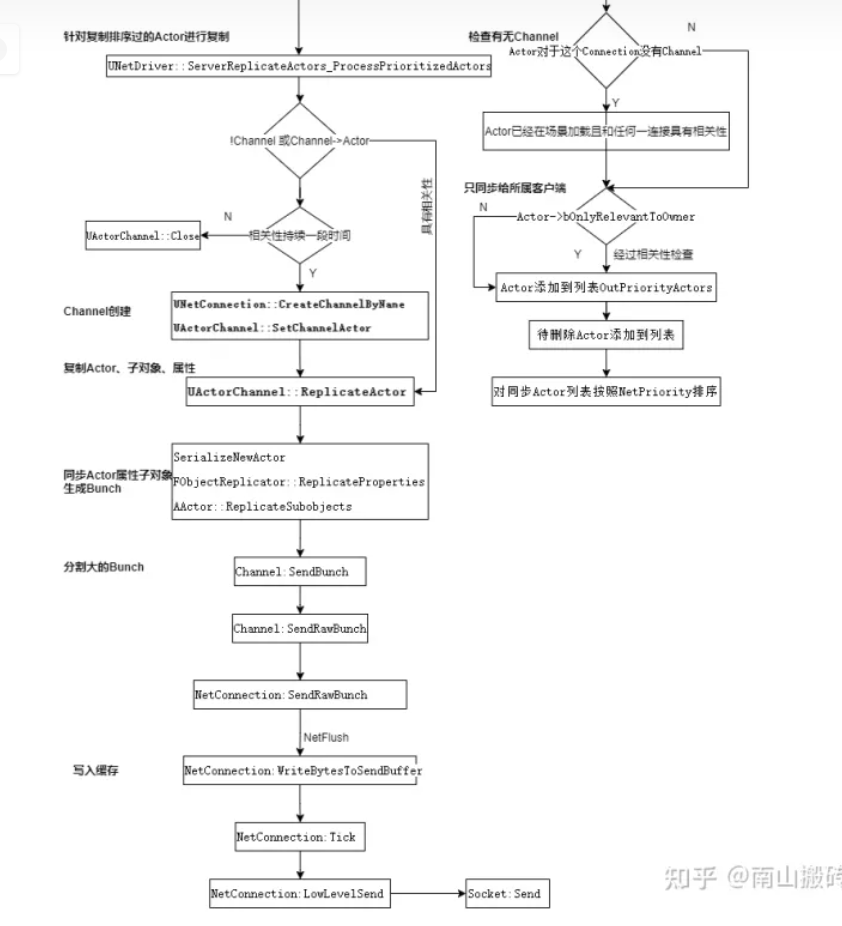
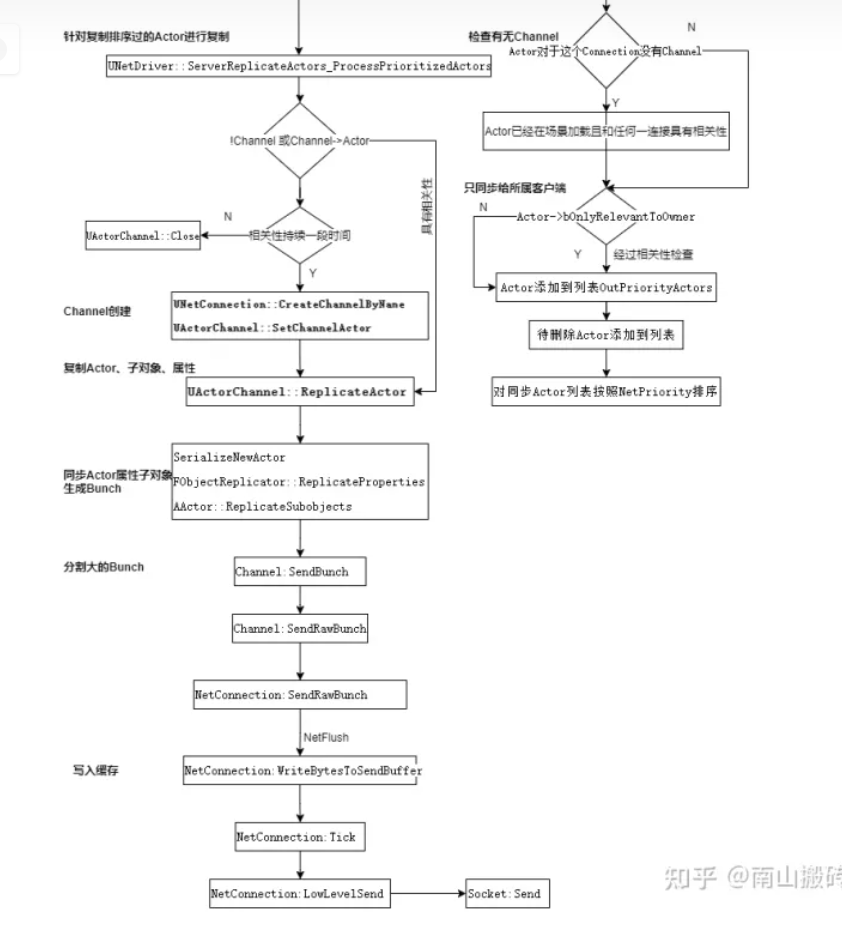
Server和客戶端连接流程



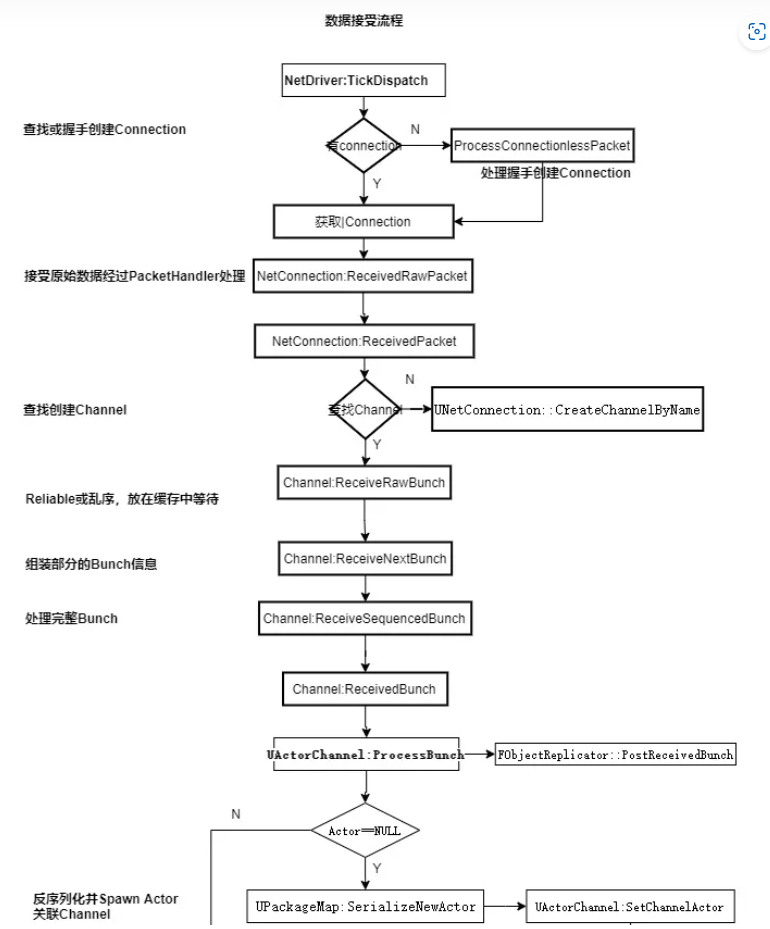
四、同步流程

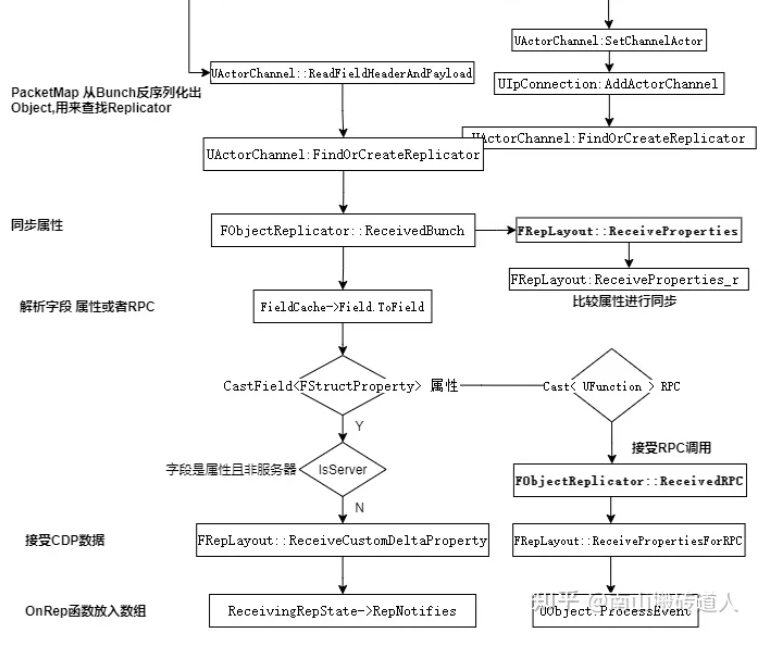
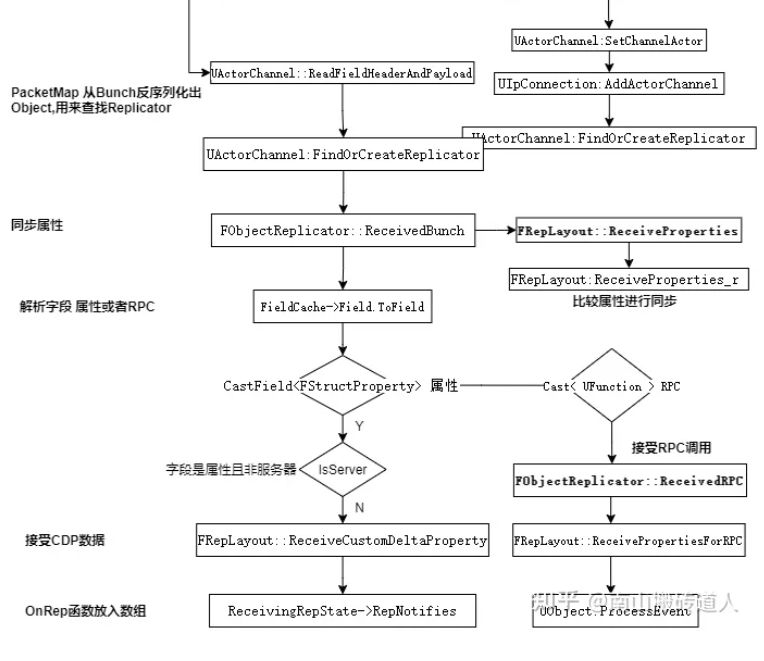
服务器总体同步流程：



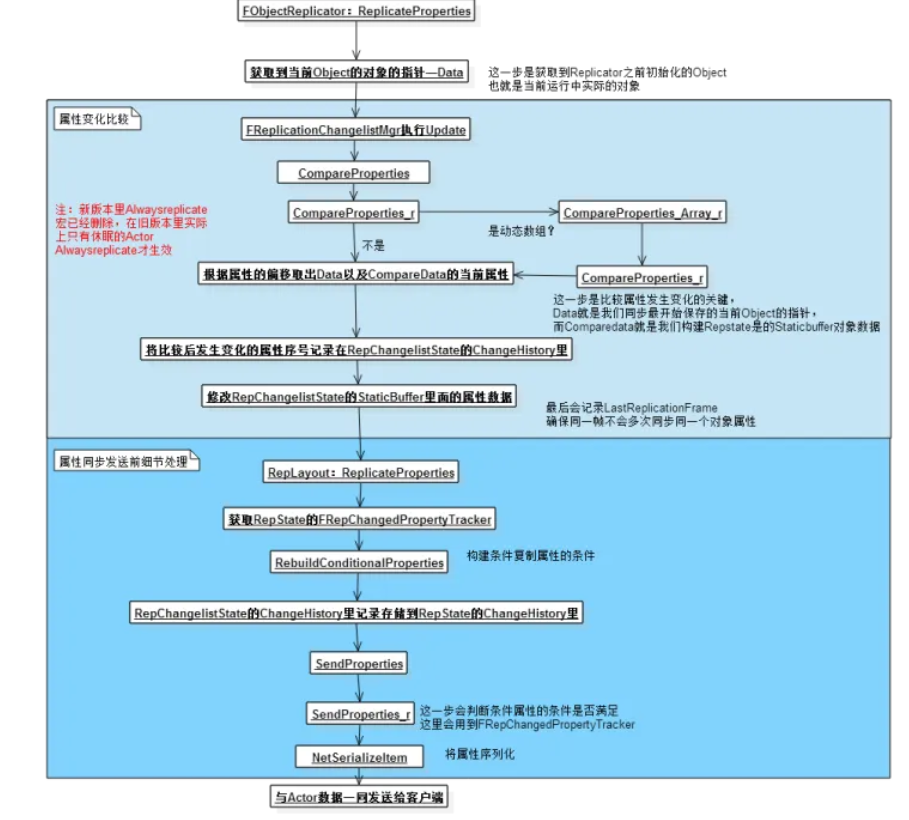
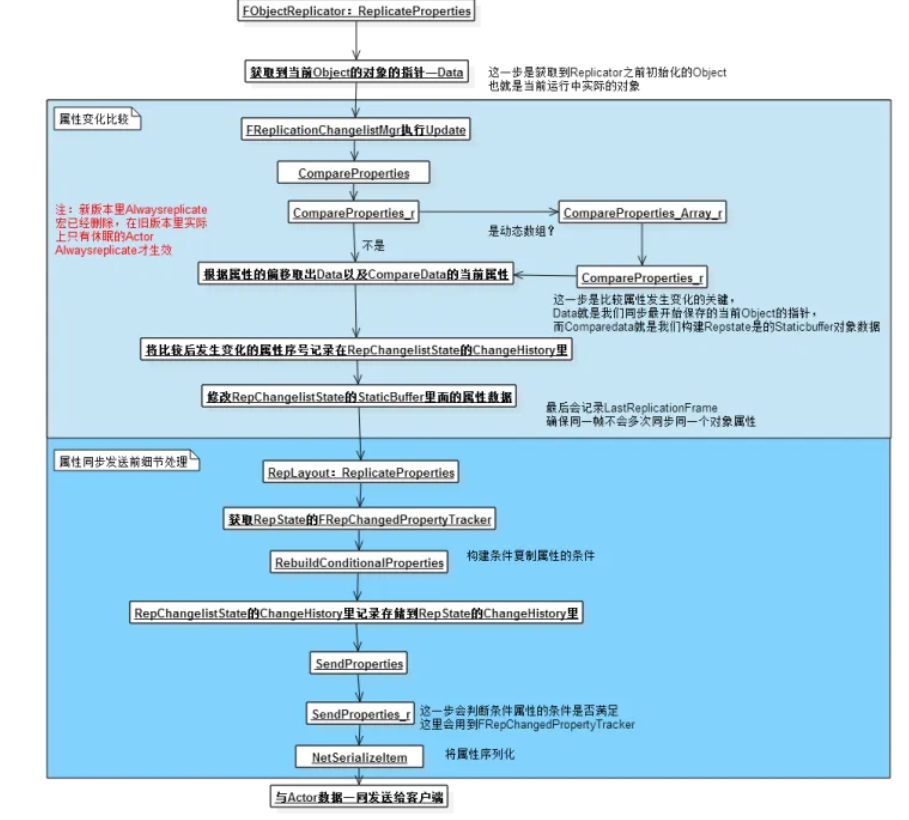


收消息流程：





4.1 关于属性的同步



说明：每帧都会进行ActorReplicate，并且比较属性，存在属性的修改的话就记录修改，并且序列化后与Actor数据一同发送到客户端，客户端接受到后反序列化进行处理

**UHT对UPROPERTY(Replicated)宏处理**

ENetFields\_Preivate 为所有replicated属性定义了一个编号，称为RepIndex， 之后属性同步时，会用RepIndex关联的CmdIndex标识传输的属性， 。每个类的ENetFileds并不是从0开始计数，而是从基类的最后一个ENetFields enum开始，因为继承自基类，自然也要继承基类的属性同步

**GetLifetimeReplicatedProps函数**

GetLifetimeReplicatedProps函数用于向引擎注册一个类的网络同步属性，在为Actor创建Channel时，会调用到该函数。

**RepLayout**

RepLayout是UE4 runtime实现属性同步的核心，有管理属性历史值，比较属性，发送属性数据等功能。

管理属性历史值：当属性发生改变时才会同步，因此引擎需要保存属性的历史值，才能进行比较，发现改变。

比较属性：比较两个值，判断属性是否改变。

发送属性数据：属性可以设置同步Condition，有些属性只需同步给特定客户端。并且属性是差量同步的，因此需要管理客户端返回的ack和nak，判断当前要发送哪些值。

**FRepLayout类结构**

该类用于维护某一类型的同步属性，可以是UClass,UStruct或者UFunction，对于一种类型，只会有一个对应的FReplayout实例，类型的多个实例共享

int32 ShadowDataBufferSize：创建一个Shadow buffer实例需要的内存

TArray<FRepParentCmd> Parents：顶层layout Command

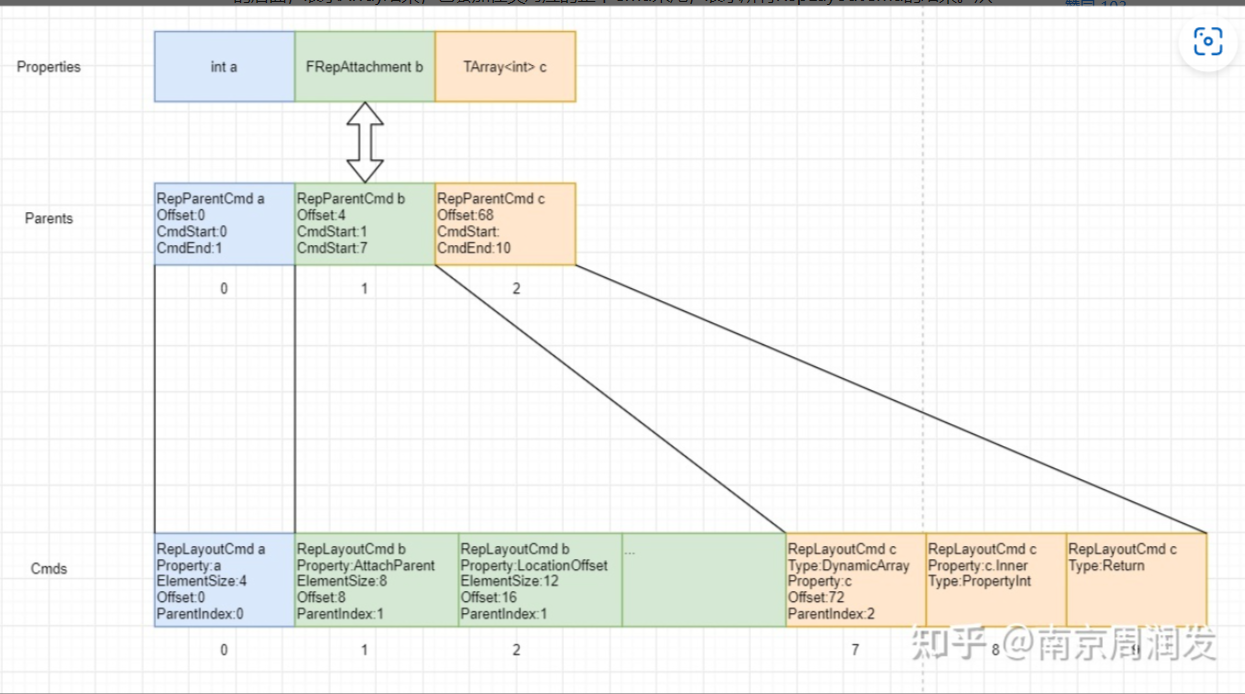
TArray<FRepLayoutCmd> Cmds：所有layout Command

**RepParentCmd**

Cmd用于知道一个Property如何同步， 普通Property，如int和bool，RepParentCmd会关联一个RepLayoutCmd，用于描述同步细节， 而复杂的Property，如TArray和UStruct，有多个RepLayoutCmd

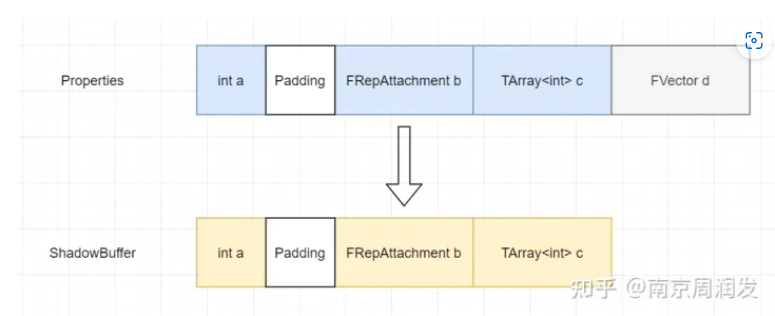
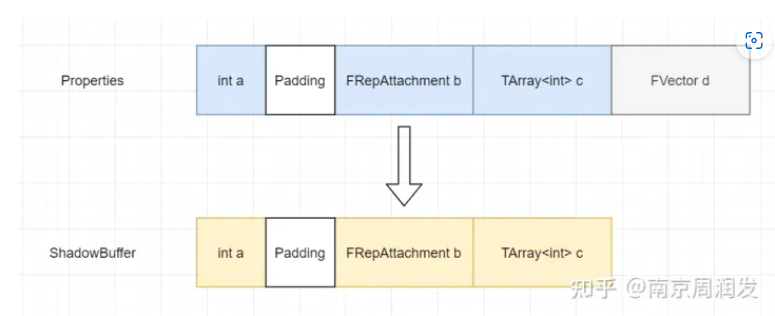
UE4将反射信息存到FProperty中，包含上述信息，所以结构臃肿，处理同步需要去遍历一遍UClass的FProperty链接，会产生性能浪费，且 FProperty用链表组织，缓存不友好，因此UE用了专门的Cmd数组记录

属性同步信息，精简且缓存友好



**ShadowMemory**

当属性改变时，才需要执行同步，一种方式是改变属性后触发回调函数，但是改变属性操作很高频，会严重影响效率，因此UE采用的方式是： 存储了一份同步属性的历史副本，会间隔一段时机把属性最新值和副本比较，从而发现哪些属性发生了改变。整个副本称为ShadowMemory。引擎代码中对应实现的变量名为ShadowBuffer。 每个Actor实例都有自己的ShadowBuffer，因为属性各自独立。 ShadowBuffer大小为所有Replicated属性之和，在加上一些内存对齐消耗，可以认为是Actor属性的一个子集。运行时，只遍历Repilcated属性，然后去ShadowBuffer中找历史值比较即可。



\*创建&初始化

当NetDriver要同步Actor时，会检查该Class是否有RepLayout，没有则调用FRepLayout::CreateFromClass函数创建。

创建后，再调用FRepLayout::InitFromClass函数进行初始化。

void FRepLayout::InitFromClass(

UClass\* InObjectClass,

const UNetConnection\* ServerConnection,

const ECreateRepLayoutFlags CreateFlags)

{

\* 首先对于InObjectClass的FProperty，要设置它们的RepIndex值，与ENetFields对应。之后初始化Class的ClassReps属性，其中存储了从基类到当前类的所有要同步Property和它们的ArrayIndex，后者一般都为0。

\* 之后对于ClassReps中每一个Property，要在Parents中添加一条RepParentCmd记录，并记下下标ParentHandle。然后根据Property类型，向Cmds数组添加Cmd实例，ArrayProperty和StructProperty需要特殊处理，添加多个子Cmd，其余普通属性则直接向Cmds数组添加一个Cmd实例即可。添加的第一个子Cmd下标设置为RepParentCmd.CmdStart，最后一个子Cmd下标设置为RepParentCmd.CmdEnd。

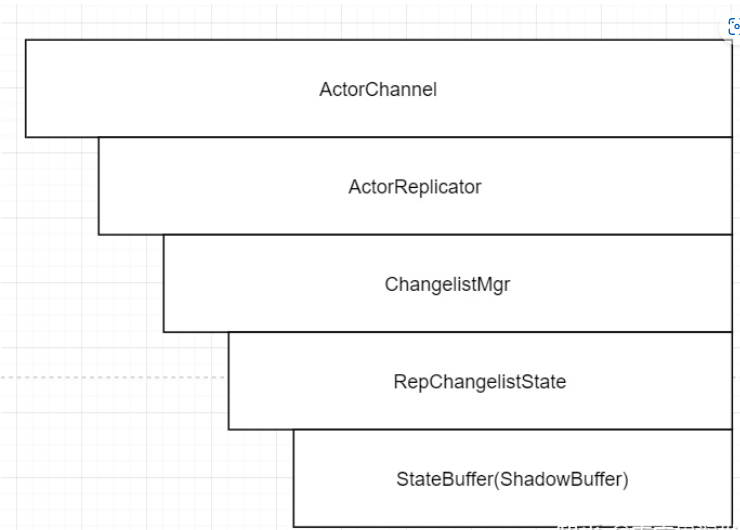
\* 处理完所有ClassReps，会向Cmds数组中再添加一个ERepLayoutCmdType::Return Cmd，表示该UClass的Property处理结束。

\* 之后调用前文提到的GetLifetimeReplicatedProps函数，得到FLifetimeProperty数组，然后用它来初始化Parents数组，设置Condition，RepNotifyCondition，RepNotifyNumParams等属性。

\* 对于Actor.RemoteRole属性，需要特殊处理，RemoteRole虽然会同步，但server和client的值不同。此时会把该属性在Parents数组中对应对应的RepParentCmd.Condition设成COND\_None，并设置ERepParentFlags::IsConditional标记。Remote和Role的实现原理在之后会介绍。

\* 执行BuildHandleToCmdIndexTable\_r函数，建立Handle到Cmd数组的映射，主要因为Array需要特殊处理。

\* 执行BuildShadowOffsets函数，计算各Cmd属性在ShadowBuffer中的偏移，记录在ShadowOffset属性中。 至此，该Class对应的Replayout已初始化完成。此时还未创建ShadowBuffer，注意Replayout 和类型一一对应，ShadowBuffer和实例一一对应。



ActorChannel: 负责传输Actor及Component的属性和RPC，创建一个Actor连接并维护连接。

ActorReplicator: 真正负责处理属性同步和rpc。

ChangelistMgr: 维护了属性的变动历史，用于确定要属性同步要发送的内容。

RepChangelistState: 实际存储属性变动历史

StaticBuffer：ShadowBuffer

**服务器同步属性**

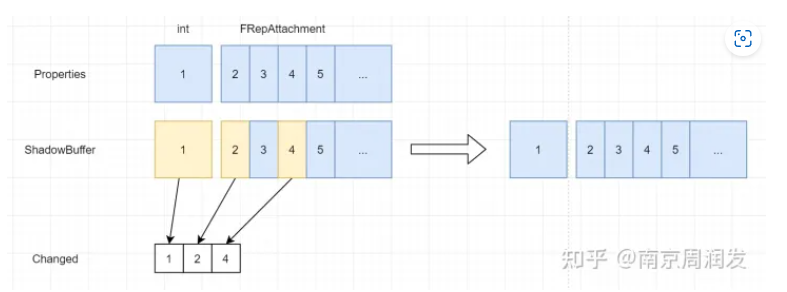
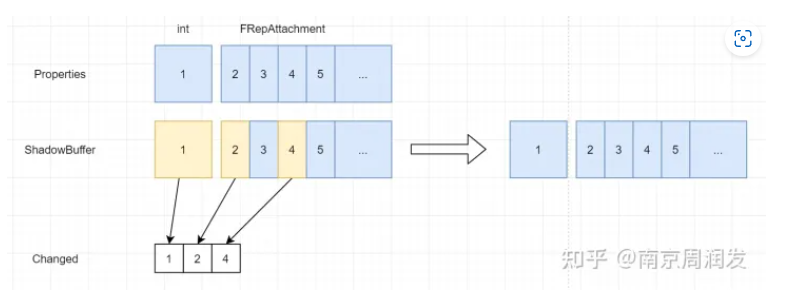
CompareProperties

首先从环形buffer RepChangelistState.ChangeHistory中取HistoryEnd下标元素，作为当前可用的Changelist，由此可见HistoryEnd总是代表一个空闲changelist， 然后调用CompareParentProperties函数做属性细节比较，最终会填充changelist的Changed属性，把Cmd的Handle写入。调用结束后，在Changed末尾添加0，作为终结符，注意Handle是从1开始计数的。 之后HistoryEnd再增长一位，如果这导致环形buffer满了，则要合并最早的两个Changelist来产生一个新的空闲，称为MergeChangeList。这个步骤和Changed内部结构有关，等分析完CompareProperties\_r后再做介绍。

CompareProperties\_r

对于每个ParentCmd，需要做详细、底层的属性比较，然后更新StaticBuffer并把Handle写入Changed数组，由CompareProperties\_r函数完成

根据属性的不同可能会采取不同的比较方式



MergeChangeList

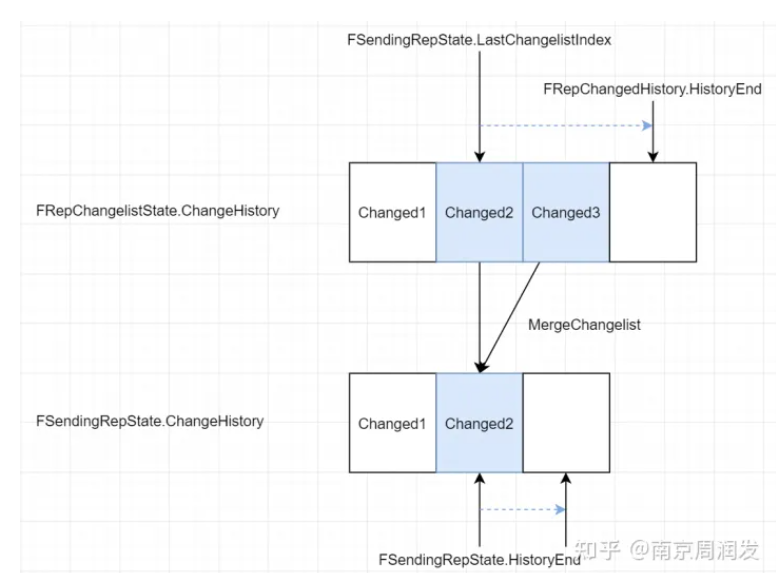
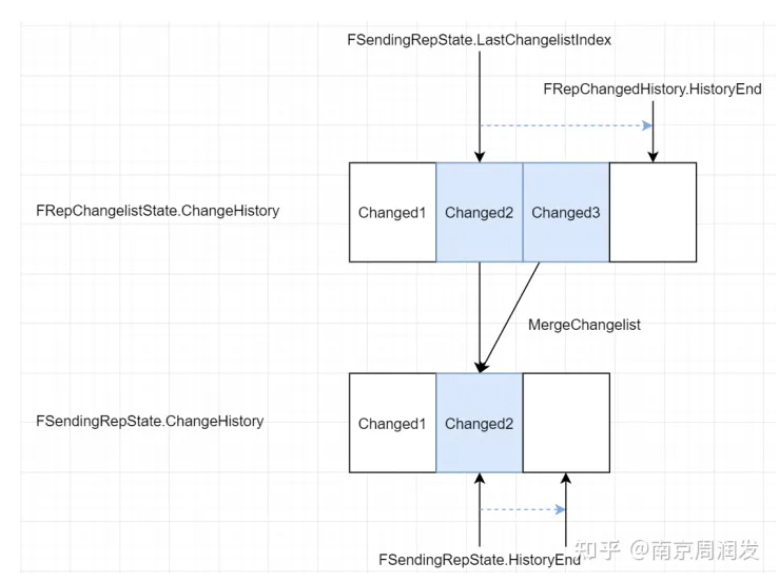
当生成新的Changelist后，如果ChangeHistory满了，就要合并最早的两个Changelist，称为MergeChangeList。

属性没有发生改变时，得到的Changed数组就为空，不会把他加到ChangeList中，于是在CompareProperties函数中直接返回了。

**服务器发送属性**

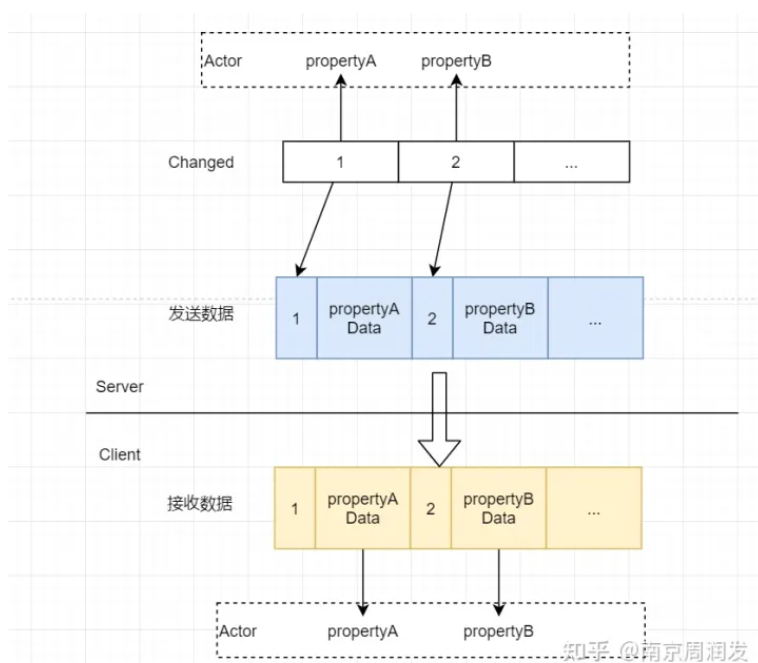
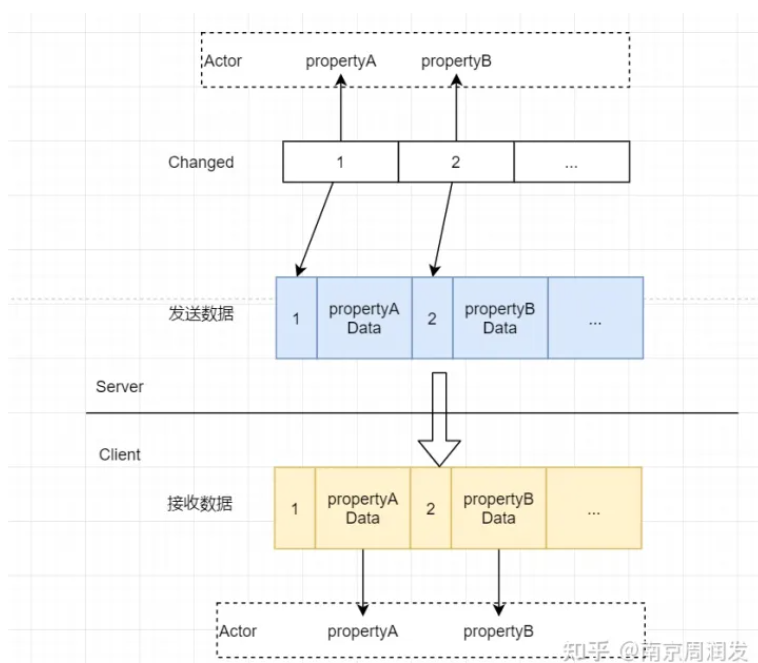
当完成属性比较，记录变化的属性后，就要发送属性同步数据了，由FRepLayout::ReplicateProperties函数完成。

FSendingRepState.LastChangelistIndex记录了最近从FRepChangelistState.Changelist中同步Changed的下标，而FRepChangelistState.HistoryEnd记录了Changelist中最新Changed下标。

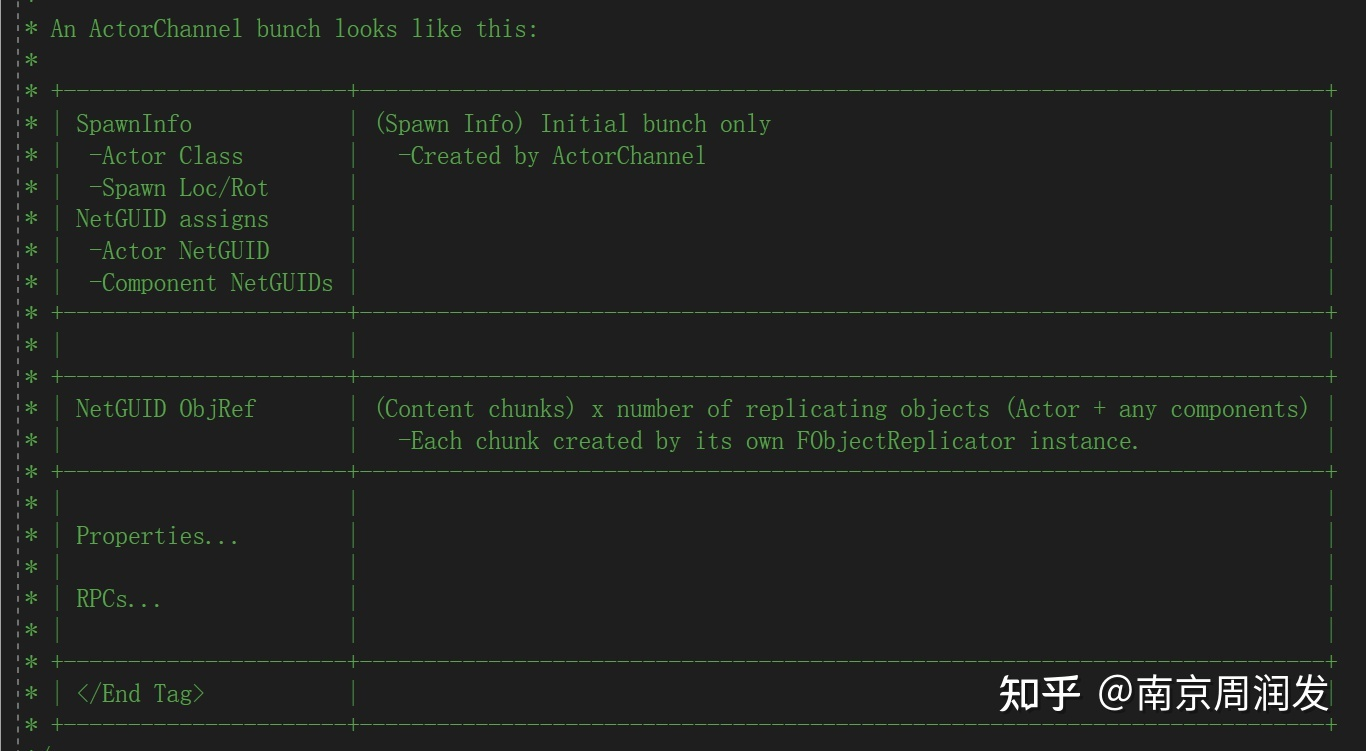
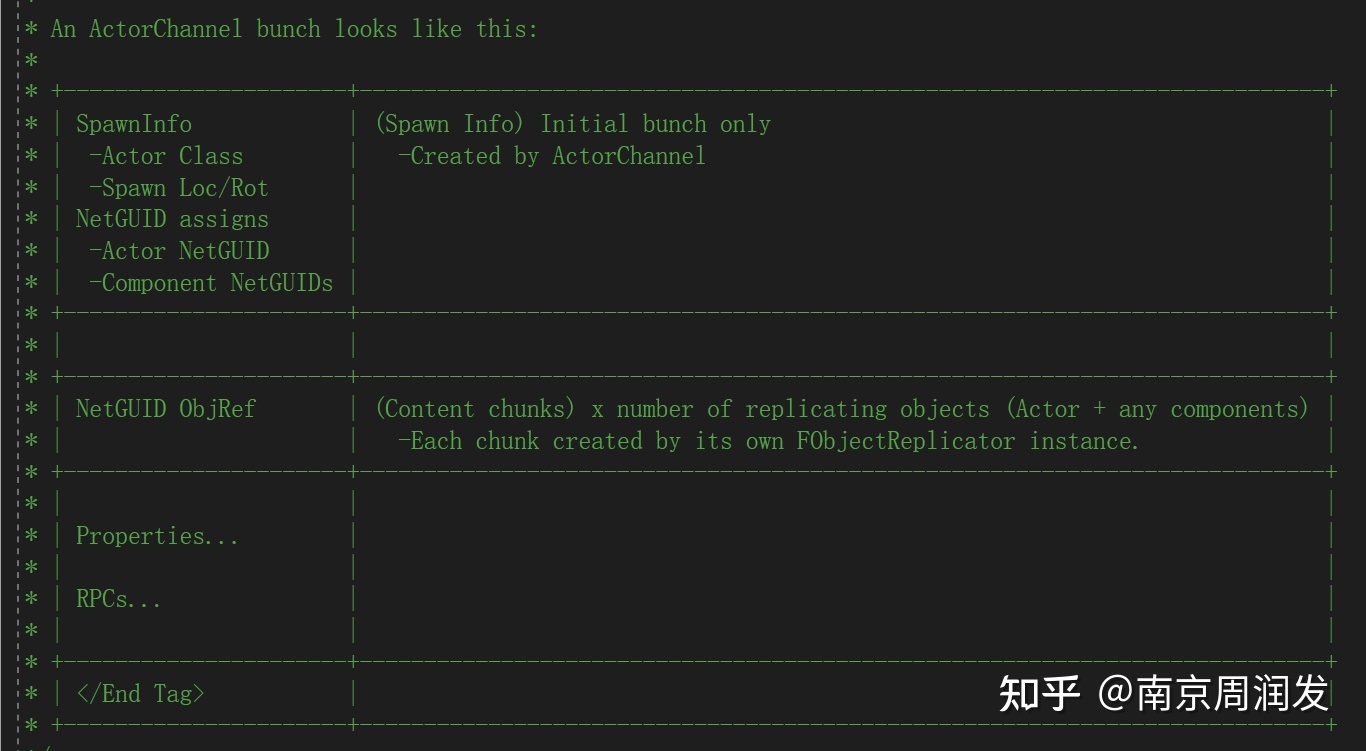


发送数据

我们知道了改变属性对应的Handle列表，接下来需要根据这些Handle找到对应属性，然后序列化属性内容，通过FRepLayout::SendProperties函数实现。 属性对应的Handle就能作为属性的标识，可以关联到ParentCmd，ReplayoutIndex等内容。因此UE发送的一个基础属性同步数据由Handle+属性内容构成。



**客户端接受属性**

****

Client使用FInBunch数据结构来描述客户端收到的Bunch信息。

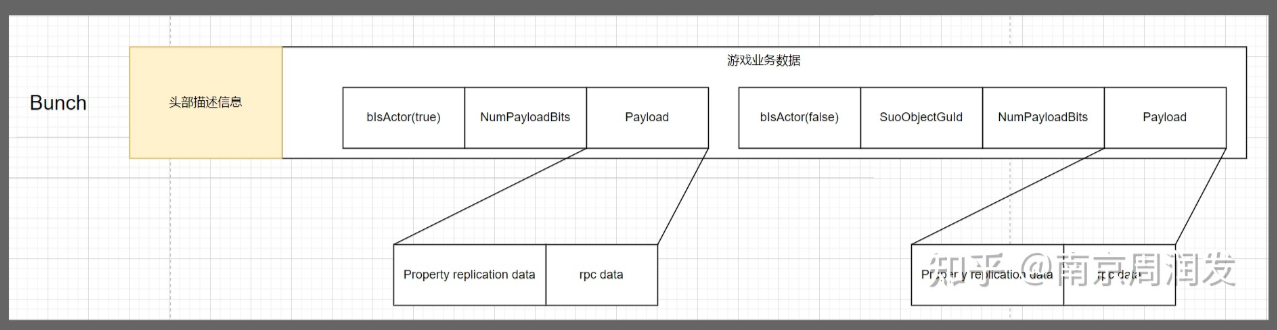
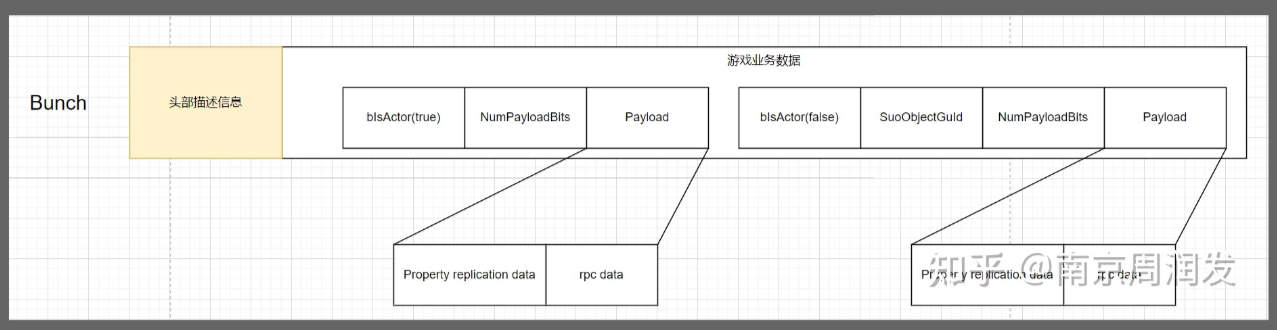
ActorChannel

其用于在网络上交换Actor/SubObjects(ActorComponent)的属性和rpc数据，因此我们可以在ActorComponent上使用属性同步和rpc，一个ActorChannel对应一个Actor。

主要属性

TObjectPtr<AActor> Actor：对应Actor

FNetworkGUID ActorNetGUID：Actor的网络Id



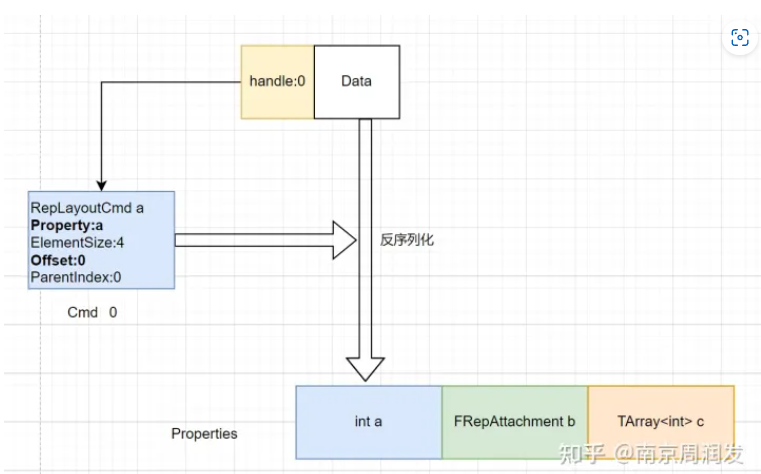
FRepLayout::ReceiveProperties

客户端逻辑相对简单，只要不断读取handle和数据即可。handle为变长int，使用SerializeIntPacked读取，接着使用ReceiveProperties\_r函数读取属性数据。

与发送一样，属性分普通属性和DynamicArray两种。

反序列化属性同步数据的核心为根据handle找到RepLayoutCmd，再通过RepLayoutCmd找到属性的FProperty以及属性在Object中的偏移，最后使用FProperty的NetSerializeItem方法，把数据写入到对应地址。

普通属性



关键代码为：



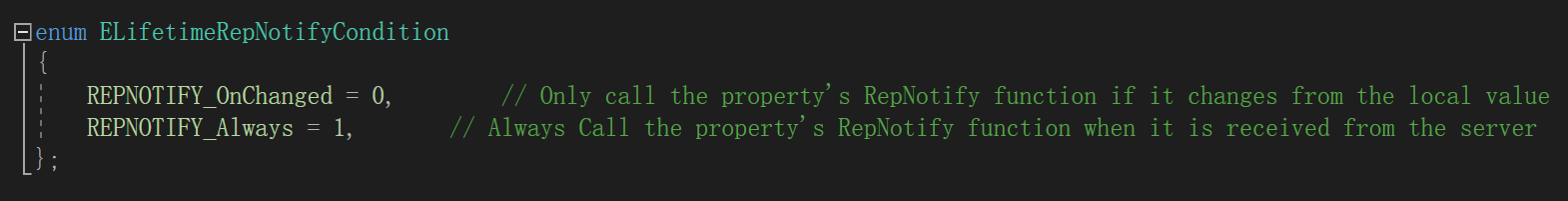
※普通属性

这里Data类型是FRepObjectDataBuffer，可以当初Object地址，重载了“+”运算符，与Cmd相加会加上对应属性的Offset，就是例子中a的地址。

如果是Struct中属性，处理方式也类似，handle可以标记到struct中每一个子属性，此时Cmd.Property就是子属性的FProperty。需要注意的是，这个FProperty中的Offset是子属性在Struct中的内存偏移，而Cmd.Offset是子属性在Object中的偏移，因此这里必须用Cmd.Offset。

**RepNotify的实现**

回调函数触发条件有两种，由ELifetimeRepNotifyCondition控制



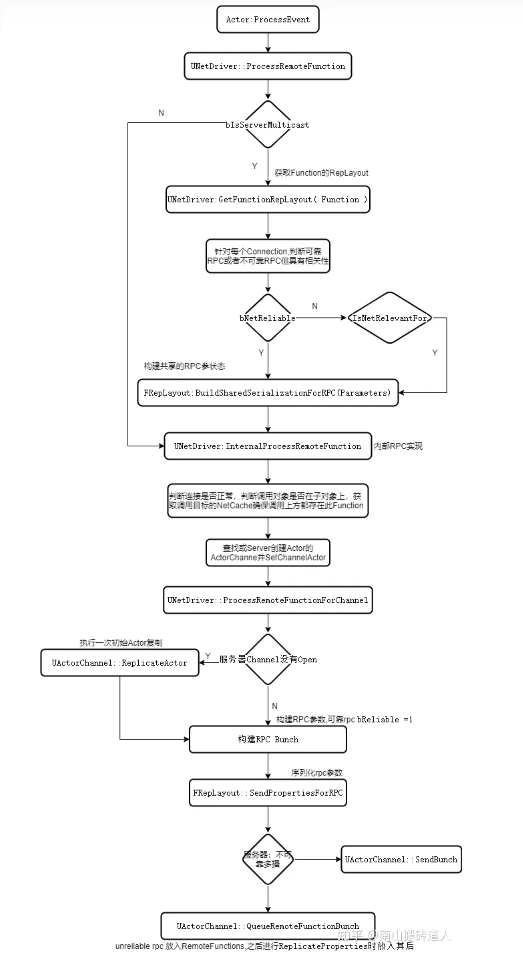
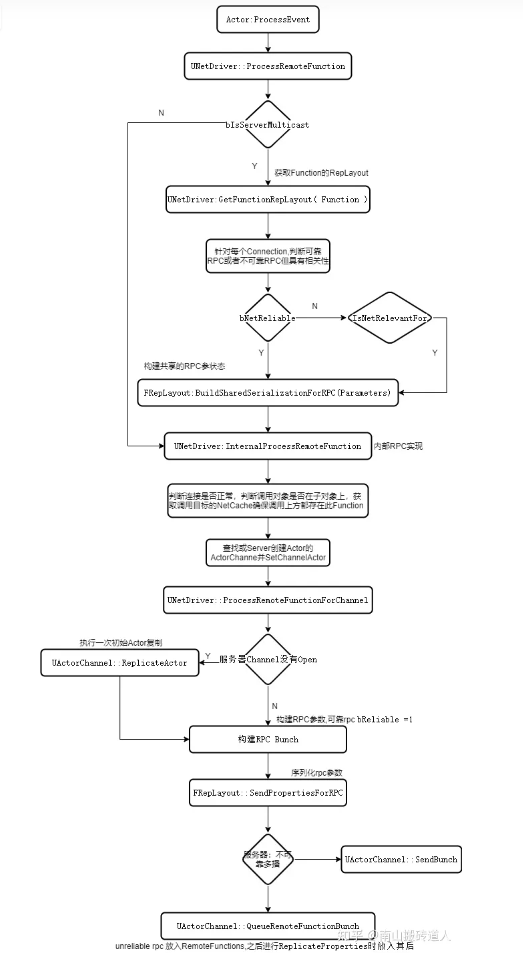
默认为改变后触发，如果本地值经修改后恰好与服务器同步下来值相同，就不会触发。还可以配置成总是触发，无论本地值是否改变。

调用RepNotify

并不是在属性改变后立即调用repnotify，而是在ReceivePropertyHelper中先记下来需要调用的属性，等ObjectReplicator处理完Bunch后，在PostReceivedBunch中再统一调用。

具体调用过程为先根据RepNotifyFunc函数名找到UFunction，再用ProcessEvent调用函数。

4.2 关于RPC的同步

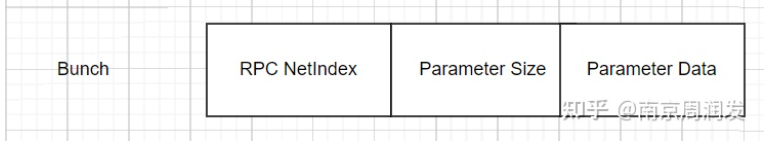
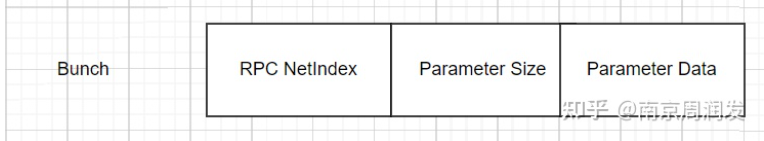


RPC流程

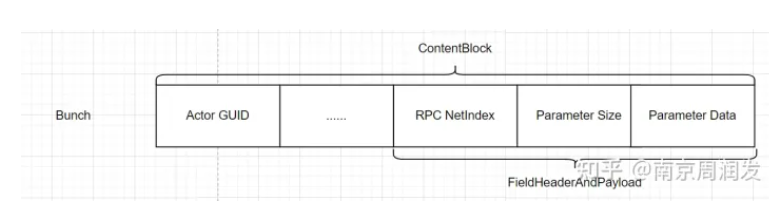
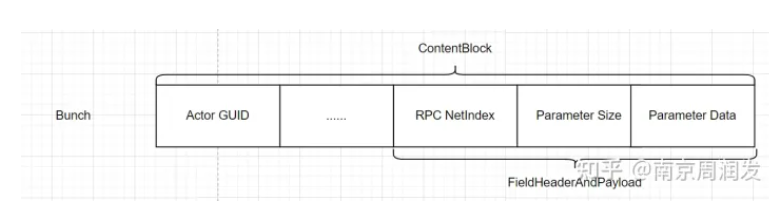
UHT会为RPC函数生成相关的带Implementation的函数声明，然后RPC函数体内会调用ProcessEvent，发现是RPC函数之后会直接交给 NetDriver::ProcessRemoteFunction处理，而不是本地执行，因为RPC本质还是通过网络发送一段数据。

先考虑单播情况，需要找到Actor所属的NetConnection，然后从Connection中找到Actor对应的ActorChannel，然后进入ProcessRemoteFunctionForChannelPrivate函数执行具体逻辑。

对于每个Actor，其中同步的属性与rpc都有一个唯一的NetIndex，发送rpc时，只需发送对应的NetInex即可，接收端能找到对应UFunction，然后再通过反射执行。rpc也可以有参数，参数使用RepLayout进行序列化，跟在NetIndex之后，这些内容会写入一个新创建的bunch，写入后，bunch内容如下：



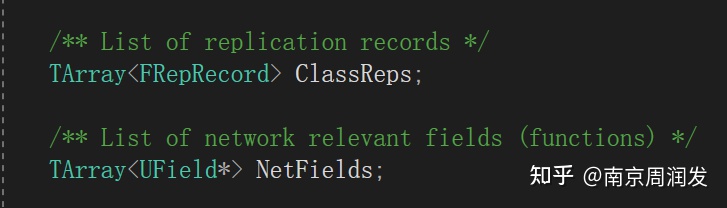
之后可以选择这个Bunch是作为单独的Bunch发送，还是等到一帧末尾与属性同步数据合并在一个Bunch里发送下去，默认只有unreliable的多播rpc才会等待合并，其他都生成单独Bunch，后面发送。这里先看单独Bunch情况。



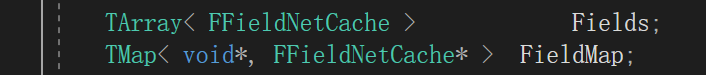
### **NetIndex分配**

rpc的一个关键点为函数Id分配，即NetIndex。

UClass在初始化网络数据时，会遍历自身的ReplicatedProperty和rpc函数，Property存储于ClassReps容器，rpc存储于NetFields容器。容器中元素会按照Name排序，顺序是固定的。



NetDriver中的NetCache属性缓存了Class的网络信息，包括Property和rpc，按照NetDriver的使用方式对信息进行了重新组织，都用FFieldNetCache类型表示它们。Fields属性存储了原始数据，FieldMap存储了映射，对于rpc，key为UFunction。NetDriver根据UClass信息初始化时，只需要先把ClassReps挨个加入Fields，再把NetFields挨个加入Fields即可，rpc的NetIndex就是其在Fields中的下标。



### **rpc参数序列化**

rpc支持参数，参数序列化与属性同步的序列化有些相似，都要使用RepLayout。UFunction的参数都会创建对应的FProperty反射数据，其中记录了属性的类型、偏移等，在使用ProcessEvent时，rpc的参数被处理为一块内存params，使用对应FProperty解析。

IMG_305

在ProcessRemoteFunctionForChannelPrivate函数中，会先为rpc创建对应的RepLayout，然后使用SendPropertiesForRPC方法，序列化参数。最后会进入SerializeProperties\_r方法，就是之前Actor属性同步的流程了。对于多播rpc，需要发送给所有客户端，因此也会缓存属性序列化后的二进制数据，这样可以避免多次序列化。

如果参数中有Uobject\*类型，只会序列化一个NetGUID，由于rpc是无状态的，只会发一次，客户端收到时如果对应UObject还没创建，这个属性就会被反序列化为NULL。

### **发送rpc时ActorChannel还没创建怎么办**

rpc会在游戏tick过程中发送，而通常Actor的同步和Channel创建要在tick末尾，因此可能出现处理rpc时Channel还未创建的情况。处理方法也不复杂，在此时立即调用一次ReplicateActor，写入Actor初始同步数据即可。

## **接收RPC流程**

客户端在收到Bunch数据后，会循环处理FieldHeaderAndPayload，发现头部NameIndex对应UFunction，就获取该UFunction，然后进入FObjectReplicator::ReceivedRPC函数，处理该rpc。

首先执行FRepLayout::ReceivePropertiesForRPC函数解析参数，写入到新建的Parms内存块中，之后继续调用ProcessEvent，执行该rpc。这里粗看流程和发送时流程相同，但发送时会调用CallRemoteFunction发送数据，并不会真正执行UFunction。区别在于FunctionCallspace的获取

IMG_306

其中会根据LoacalRole和函数的Client/Server标记得到不同Callspace，在接收方，Callspace就是Local，会调用Invoke函数，本地执行。先调用execXXX，最后再进入XXX\_Implementation函数。

## **单播、多播RPC区别**

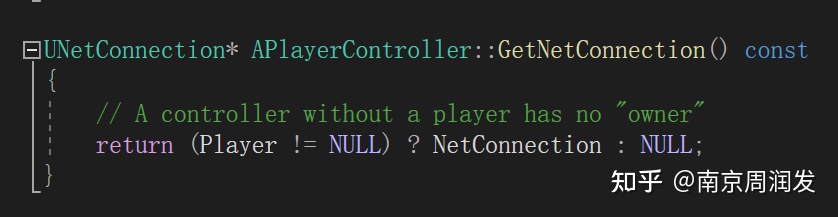
单播rpc只要发送给Actor所属的Connection，而server上的多播rpc要发送给所有ClientConnection

单播rpc和reliable多播rpc在调用函数时立即生产bunch，写入sendbuffer，而unreliable多播rpc要等到tick末尾，执行属性同步时一起处理，这个时机不一定，受同步频率，网络相关性等影响

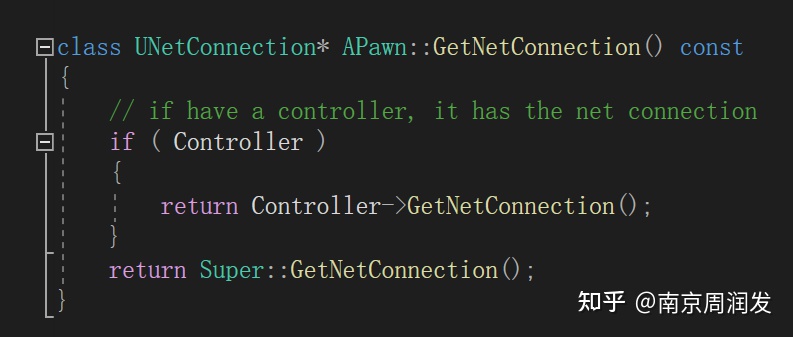
Actor所属的Connection如何确定

通过GetNetConnection可以获取，看典型的3种实现

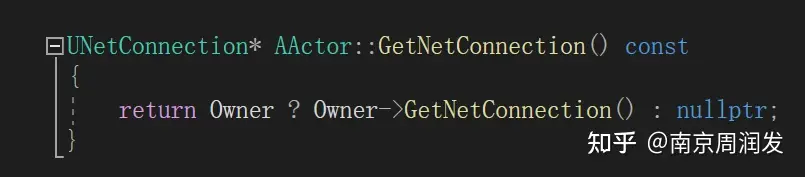
PlayerController的Connection可直接获取



Pawn的Connection为当前Controller的Connection



Actor的Connection为Owner的Connection



因此想让一个Actor能执行单播rpc，只要把Owner设置为Pawn或者PlayerController即可。

五、数据包相关

