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。

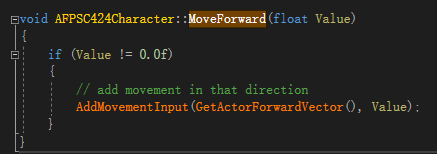
**玩家输入**

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：



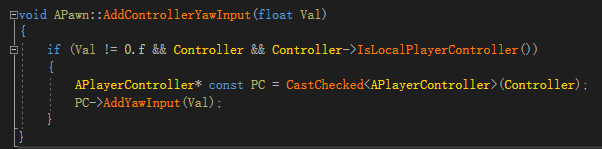
最终调用到了UPawnMovementComponent::AddInputVector函数，提供了一个向前的向量。

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。

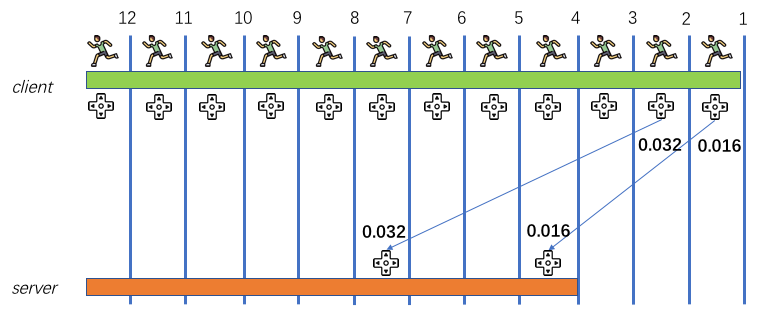
**移动时间戳**

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。

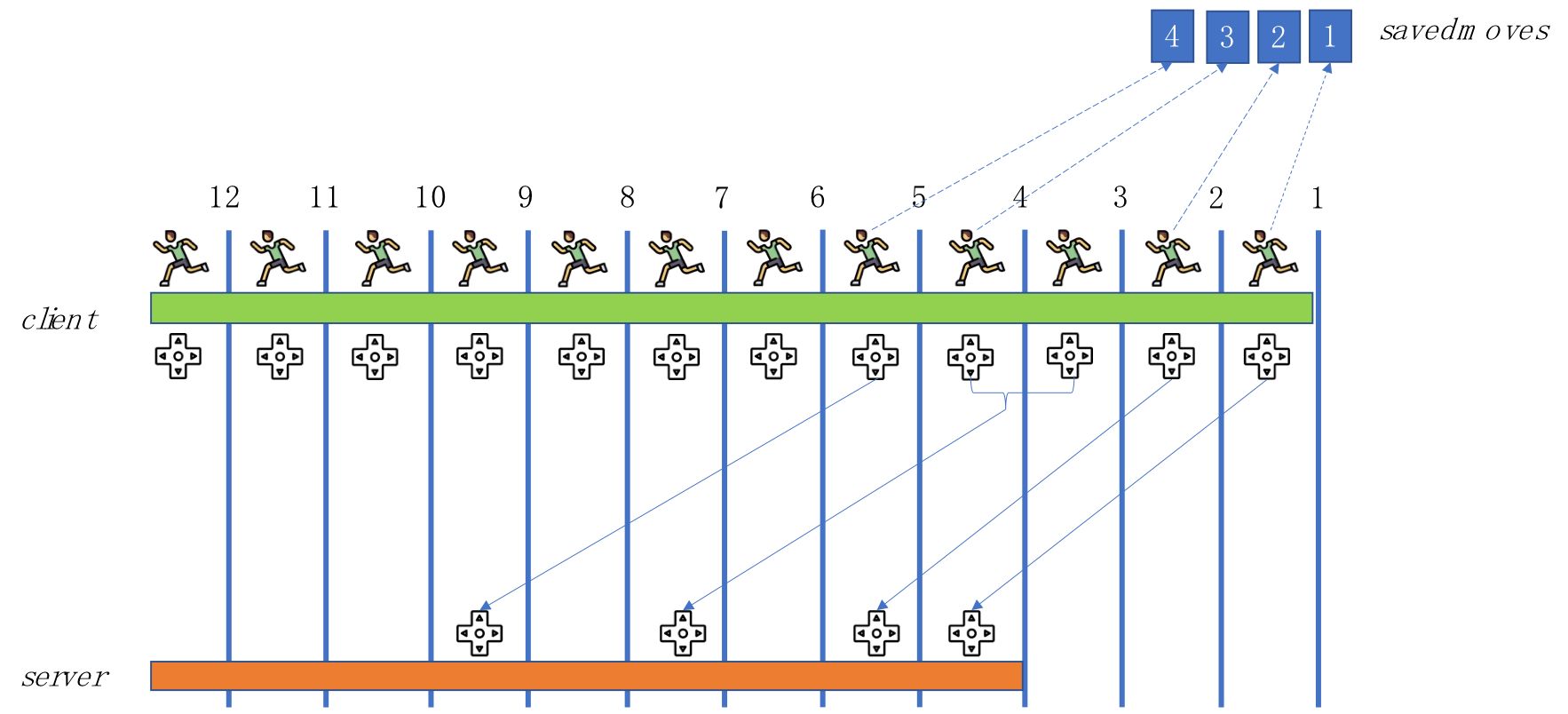
**FSavedMove\_Character缓存池**

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**

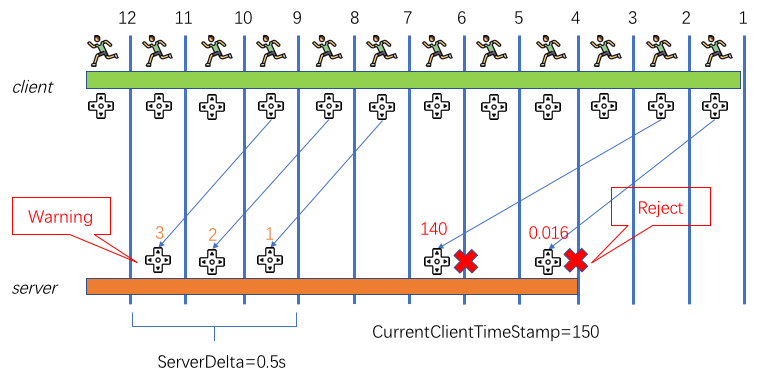
首先Server会做时间戳参数的校验。Server上对每个角色也存储了一些时间戳，在FNetworkPredictionData\_Server\_Character数据结构中有CurrentClientTimeStamp属性，表示最近处理的移动时间戳，无论移动校验是否通过，都会被更新，也会定期重置。

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**

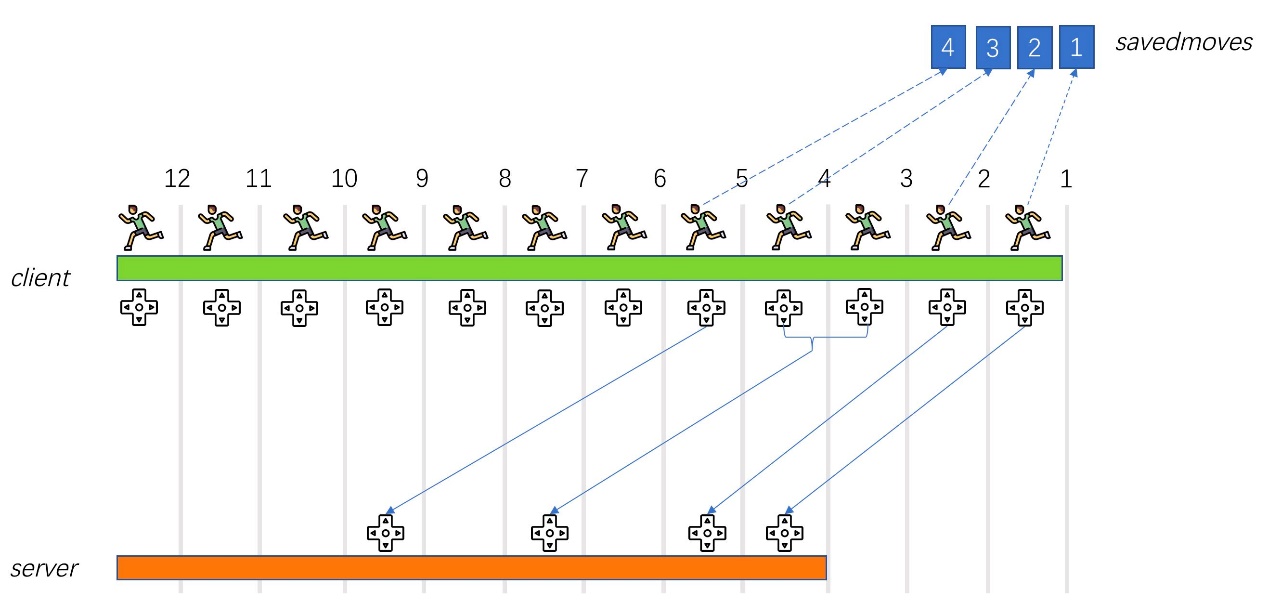
如果位置校验通过了，可以选择是否直接使用客户端位置，因为服务器位置和客户端位置相差并不大。如果服务器不用客户端位置，虽然这次检查通过了，但下次的位置误差可能会累加，导致调整。服务器使用客户端位置，会减小之后的误差，但存在作弊的可能性。引擎默认不使用客户端位置。

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。

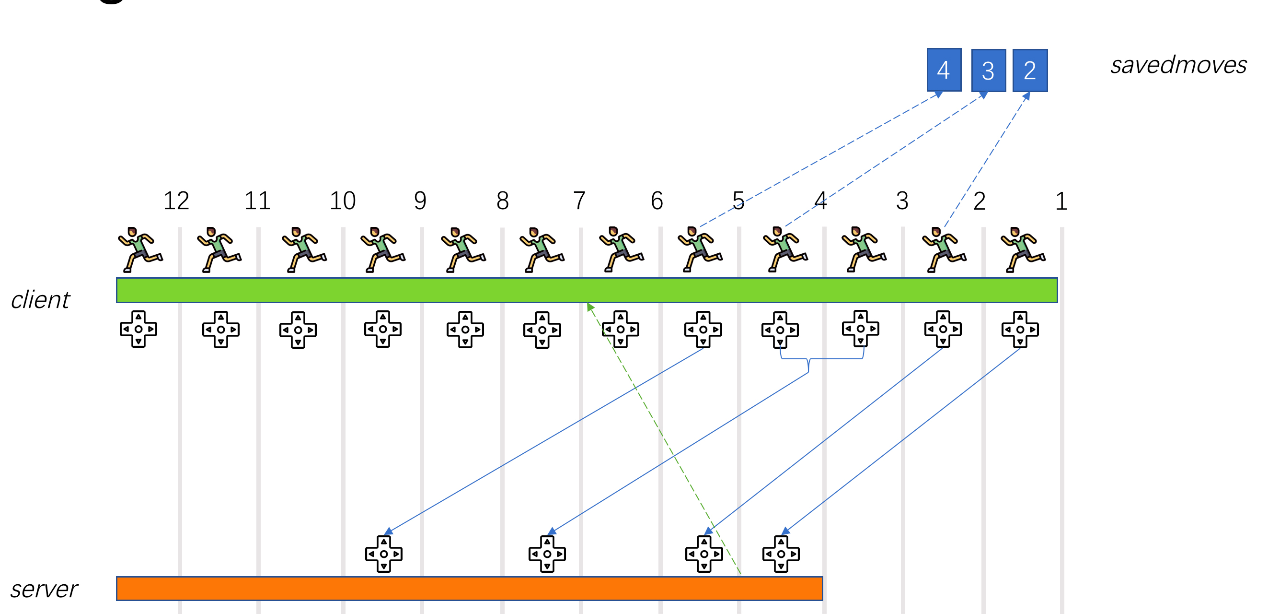
ClientAckMove的参数只有一个，就是PendingAdjustment的TimeStamp。Autonomous客户端收到rpc后，根据TimeStamp从SavedMoves数组里找到对应的move，把它作为当前的LastAckedMove，然后把SavedMoves中TimeStamp之前的Move都删除，表示之前的move都被ack了。

示例：

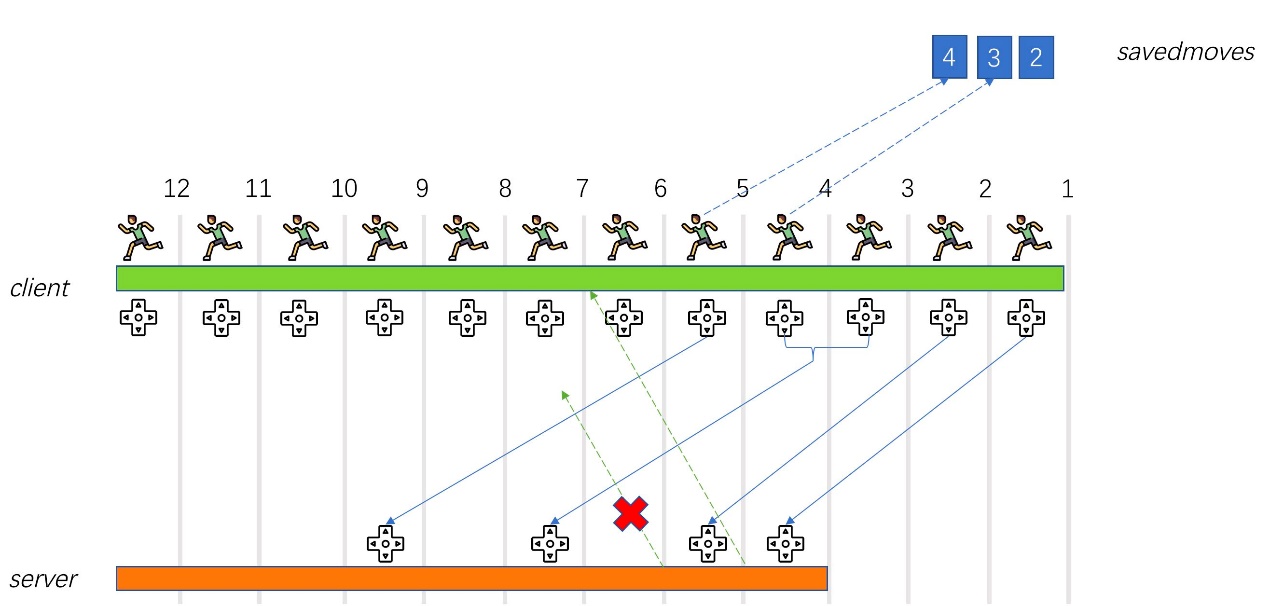
首先客户端发送了四个move给服务器



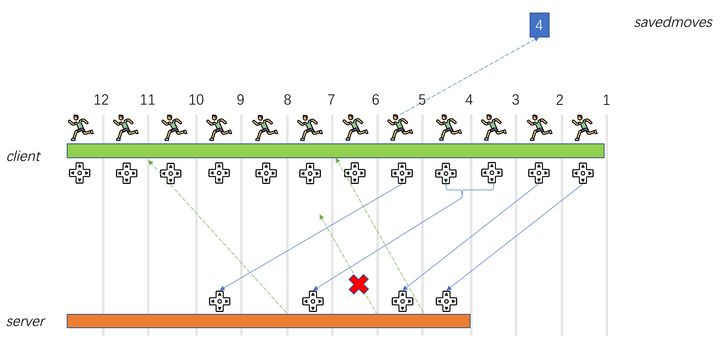
服务器针对move1发送ack给客户端，客户端收到，从savedmoves里移除move1



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。

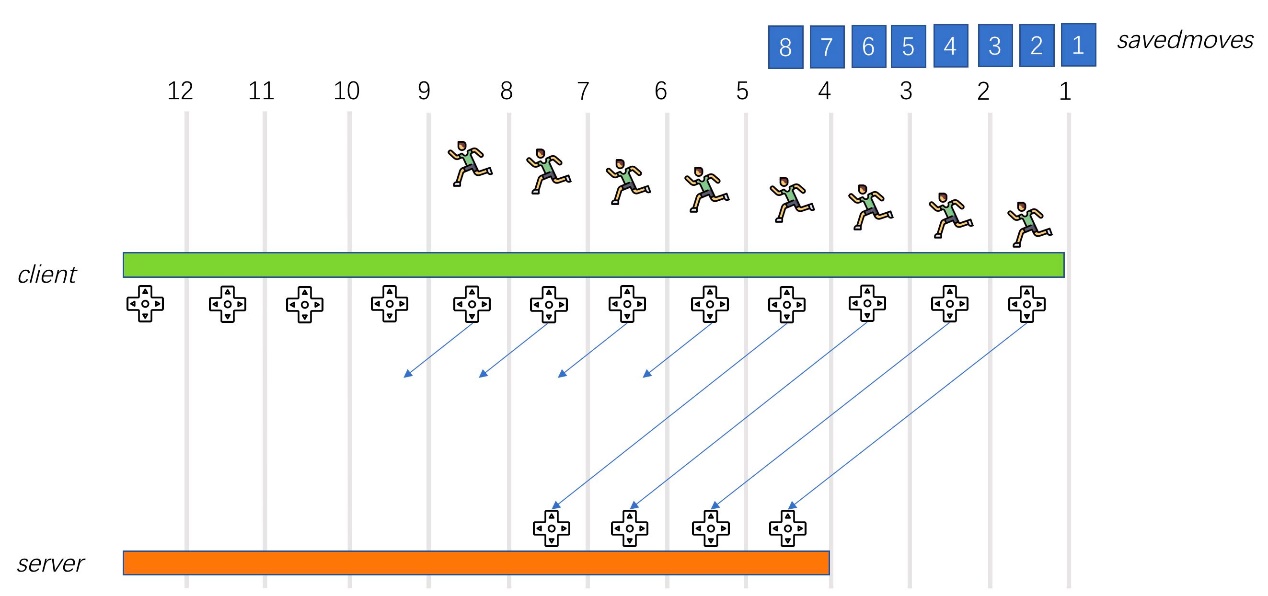
bUpdatePosition属性

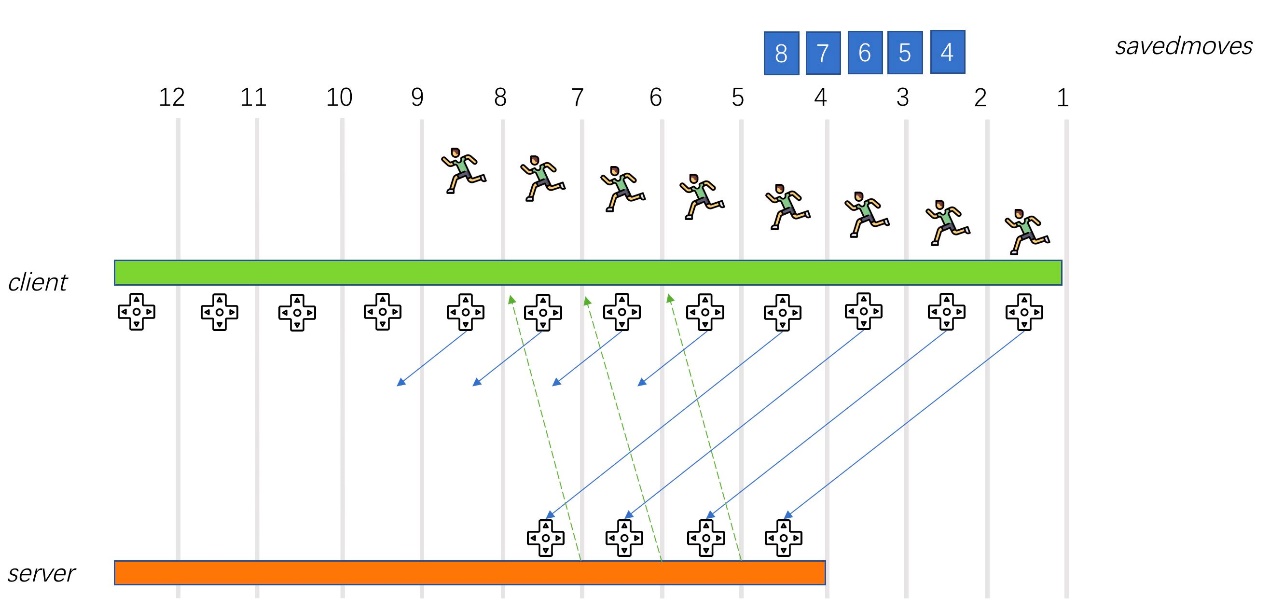
在Autonomous客户端执行TickComponent时，会检查bUpdatePosition是否为true。如果是，就要重播当前savedmoves中的所有move，做法也比较简单，先提取move信息，然后使用MoveAutonomous移动即可，这些移动不需要再发送servermove rpc了。重播结束后，玩家已经在被纠正位置的基础上把后续输入都重演了一遍，后续位置可能会和服务器算的保持一致。接着客户端继续接受输入并正常移动，可能还会收到服务器纠正位置消息，因为这些move在重播前已经发了server move rpc，这些rpc里location也是错的，服务器会继续纠正，但角色后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

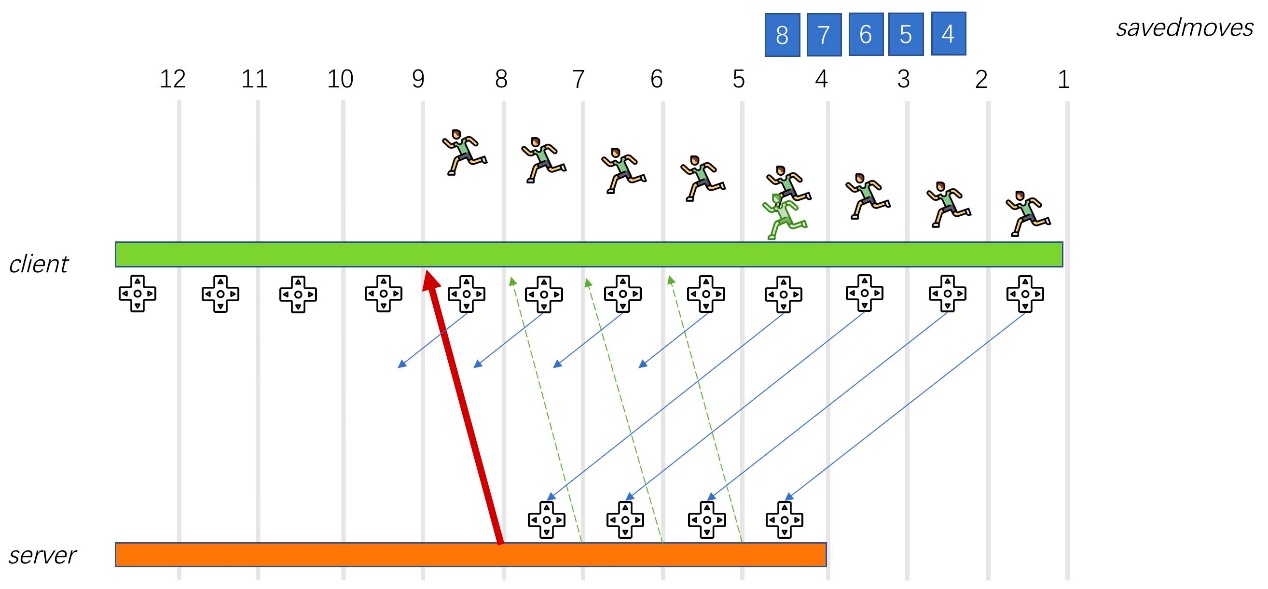
示例：

前3个move校验通过

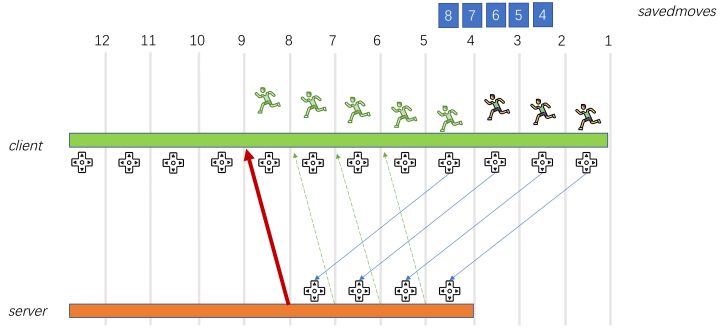




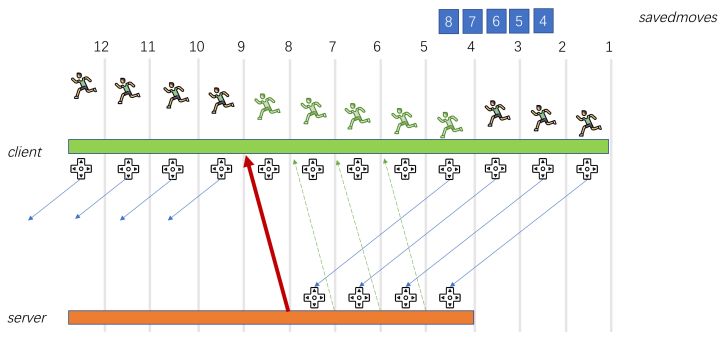
第四个位置校验不通过，服务器判定应该处于绿色位置，可能是撞墙了



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**

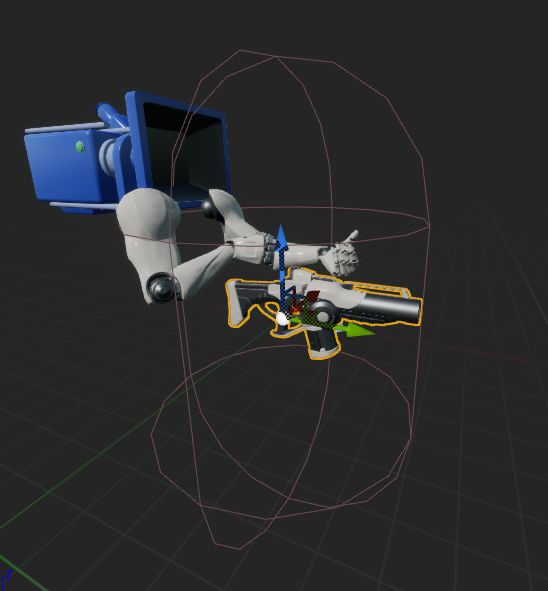
**Actor基本同步方案**

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**

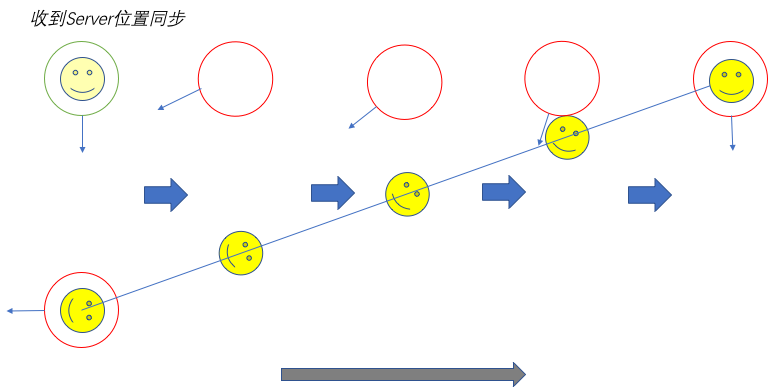
Linear使用线性插值进行平滑，视觉上位移和转向变化为匀速运动。

**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

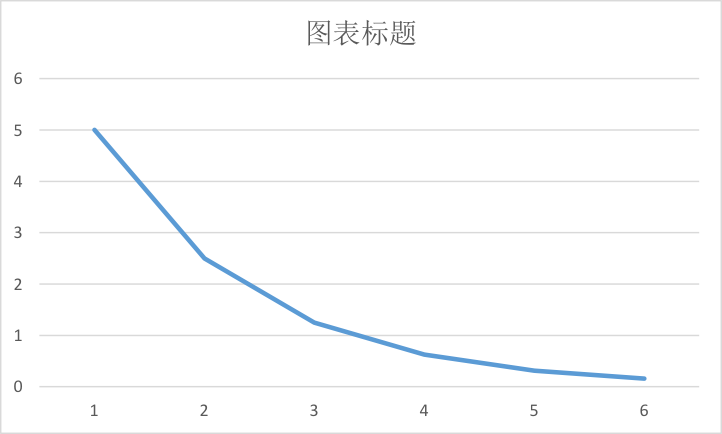
LocationOffset1 = LocationOffset0\*ΔP

LocationOffset2 = LocationOffset0\*(ΔP^2)

LocationOffsetN = LocationOffset0\*(ΔP^N)

因此称为“指数”。

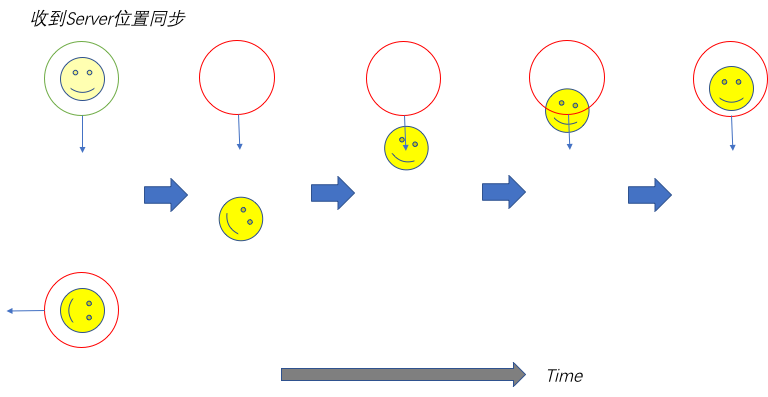
下图展示了指数插值下tick次数与位置差异的关系，可以发现是指数减小的。



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。

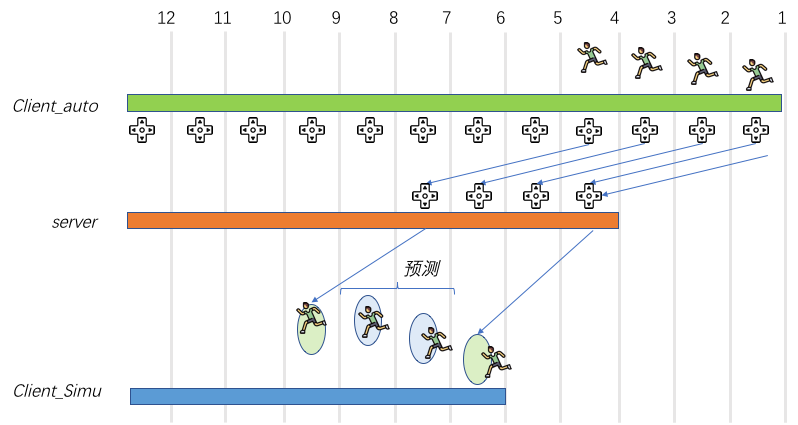


**Simulate客户端预测**

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。

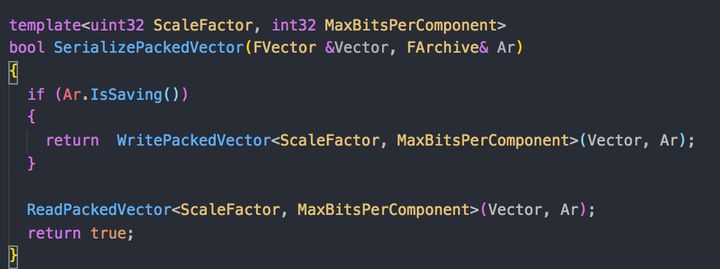
移动同步中的位置、加速度、速度都可以用FVector表示，内含3个float类型变量，总共12个Byte。相对于游戏中大部分使用场景，float的表示范围是大大溢出的，但是float遵循IEEE754标准，在网络上传输float无法使用自适应Bit流来减小数据量，导致直接传输FVector就必须用12个Byte，有些浪费。

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

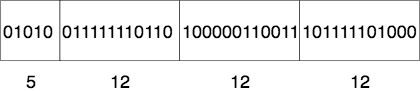
例子：使用FVector\_NetQuantize10传输(-1.0, 5.1, 100.0)

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：



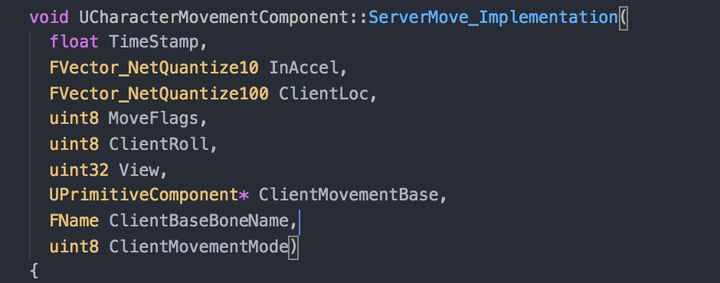
总共使用了41位，原始的三个float需要128位，仅为其32%大小。留意10和2038的第一位0填充。

**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。