参考文章

https://zhuanlan.zhihu.com/p/34257208

GravityZ：重力加速度，当重力Scale为1时，重力加速度为-980，即9.8m/s

JumpZVelocity：跳跃的初始速度，也即跳跃时的瞬间加速度。

Velocity：Actor的速度

二．移动实现的基本原理

2.1 移动组件与玩家角色

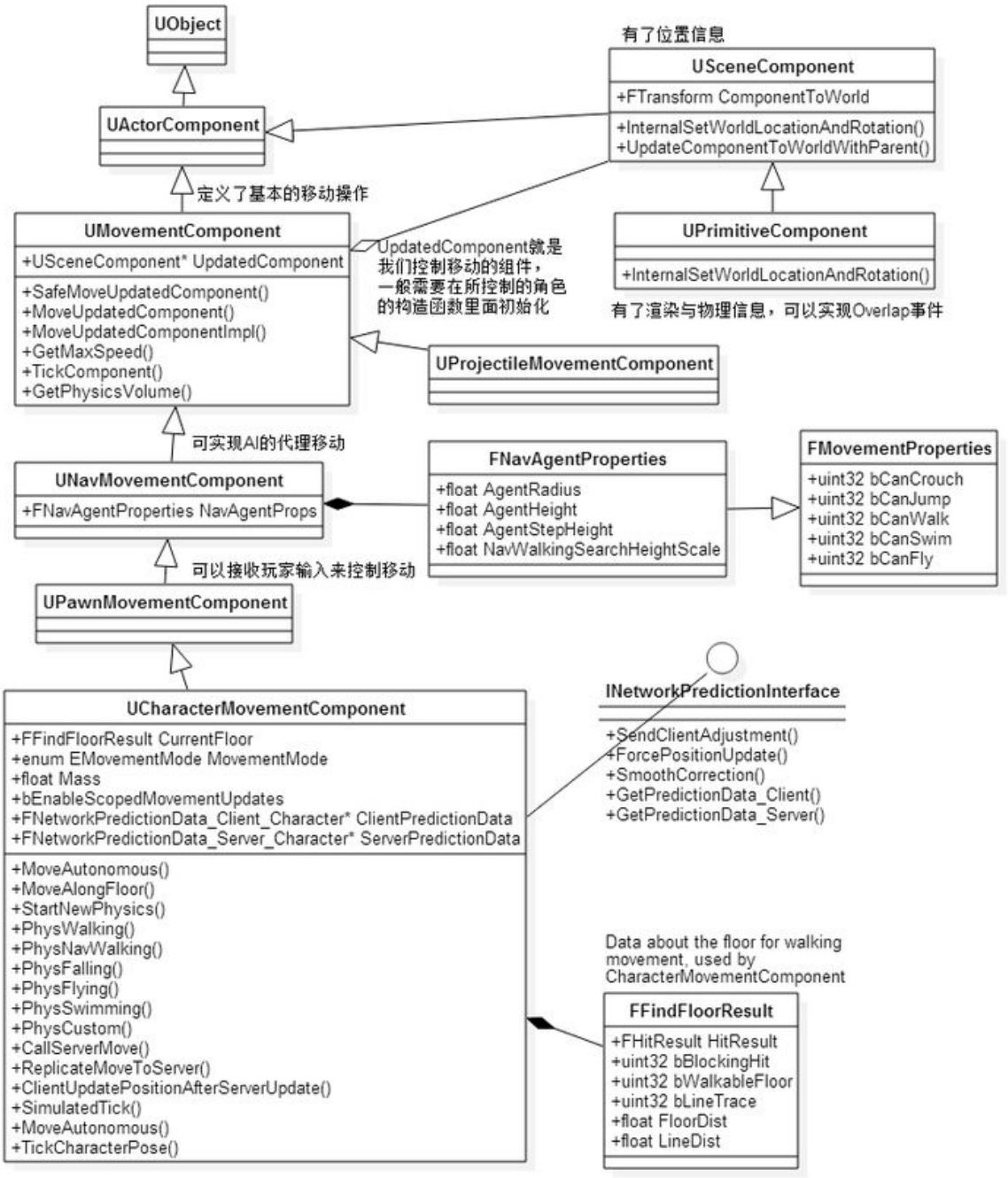
角色的移动本质上就是合理的改变坐标位置，在UE里面角色移动的本质就是修改某个特定组件的坐标位置。图2-1是我们常见的一个Character的组件构成情况，可以看到我们通常将CapsuleComponent(胶囊体)作为自己的根组件，而Character的坐标本质上就是其RootComponent的坐标，Mesh网格等其他组件都会跟随胶囊体而移动。移动组件在初始化的时候会把胶囊体设置为移动基础组件UpdateComponent，随后的操作都是在计算UpdateComponent的位置。

当然，我们也并不是一定要设置胶囊体为UpdateComponent，对于DefaultPawn（观察者）会把他的SphereComponent作为UpdateComponent，对于交通工具对象AWheeledVehicle会默认把他的Mesh网格组件作为UpdateComponent。你可以自己定义你的UpdateComponent，但是你的自定义组件必须要继承USceneComponent（换句话说就是组件得有世界坐标信息），这样他才能正常的实现其移动的逻辑。

2.2 移动组件继承树

移动组件类并不是只有一个，他通过一个继承树，逐渐扩展了移动组件的能力。从最简单的提供移动功能，到可以正确模拟不同移动状态的移动效果

移动组件继承关系图：



移动组件类一共四个。首先是UMovementComponent，作为移动组件的基类实现了基本的移动接口SafeMovementUpdatedComponent()，可以调用UpdateComponent组件的接口函数来更新其位置。

```
bool UMovementComponent::MoveUpdatedComponentImpl( const FVector& Delta, const FQuat& NewRotation, bool bSweep, FHitResult* OutHit, ETeleportType Teleport)
{
    if (UpdatedComponent)
    {
        const FVector NewDelta = ConstrainDirectionToPlane(Delta);
        return UpdatedComponent->MoveComponent(NewDelta, NewRotation, bSweep, OutHit, MoveComponentFlags, Teleport);
    }

    return false;
}
```

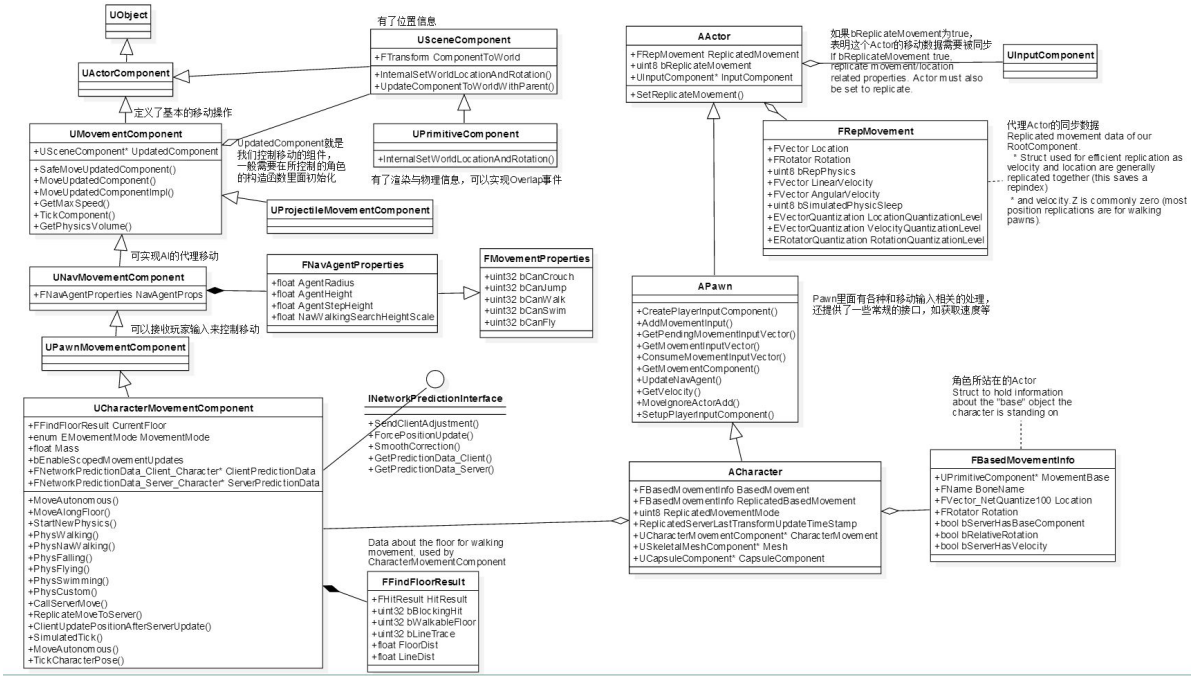
通过上图可以看到UpdateComponent的类型是UScenceComponent，UScenceComponent类型的组件提供了基本的位置信息——ComponentToWorld，同时也提供了改变自身以及其子组件的位置的接口InternalSetWorldLocationAndRotation()。而UPrimitiveComponent又继承于UScenceComponent，增加了渲染以及物理方面的信息。我们常见的Mesh组件以及胶囊体都是继承自UPrimitiveComponent，因为想要实现一个真实的移动效果，我们时刻都可能与物理世界的某一个Actor接触着，而且移动的同时还需要渲染出我们移动的动画来表现给玩家看。

下一个组件是UNavMovementComponent，该组件更多的是提供给AI寻路的能力，同时包括基本的移动状态，比如是否能游泳，是否能飞行等。

UPawnMovementComponent组件开始变得可以和玩家交互了，前面都是基本的移动接口，不手动调用根本无法实现玩家操作。UPawnMovementComponent提供了AddInputVector()，可以实现接收玩家的输入并根据输入值修改所控制Pawn的位置。要注意的是，在UE中，Pawn是一个可控制的游戏角色（也可以是被AI控制），他的移动必须与UPawnMovementComponent配合才行，所以这也是名字的由来吧。一般的操作流程是，玩家通过InputComponent组件绑定一个按键操作，然后在按键响应时调用Pawn的AddMovementInput接口，该方法进而调用移动组件的AddInputVector()，调用结束后会通过ConsumeMovementInputVector()接口消耗掉该次操作的输入数值，完成一次移动操作。

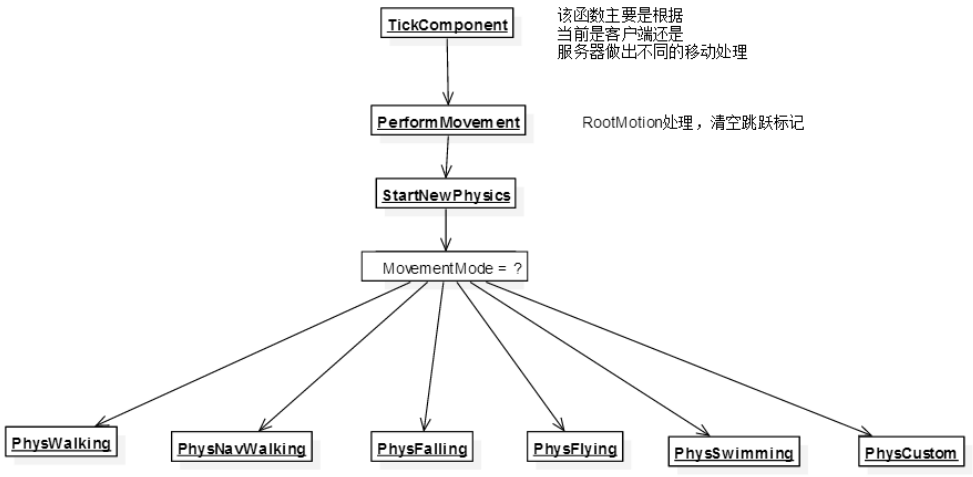
输入的调用流程：

Pawn::AddMovementInput -> PawnMovementInput::AddInputVector -> Pawn::Internal_AddMovementInput -> 修改Pawn属性



三．各个移动状态的细节处理

这一节我们把焦点集中在UCharacterMovementComponent组件上，来详细的分析一下他是如何处理各种移动状态下的玩家角色的。首先肯定是从Tick开始，每帧都要进行状态的检测与处理，状态通过一个移动模式MovementMode来区分，在合适的时候修改为正确的移动模式。移动模式默认有6种，基本常用的模式有行走、游泳、下落、飞行四种，有一种给AI代理提供的行走模式，最后还有一个自定义移动模式。

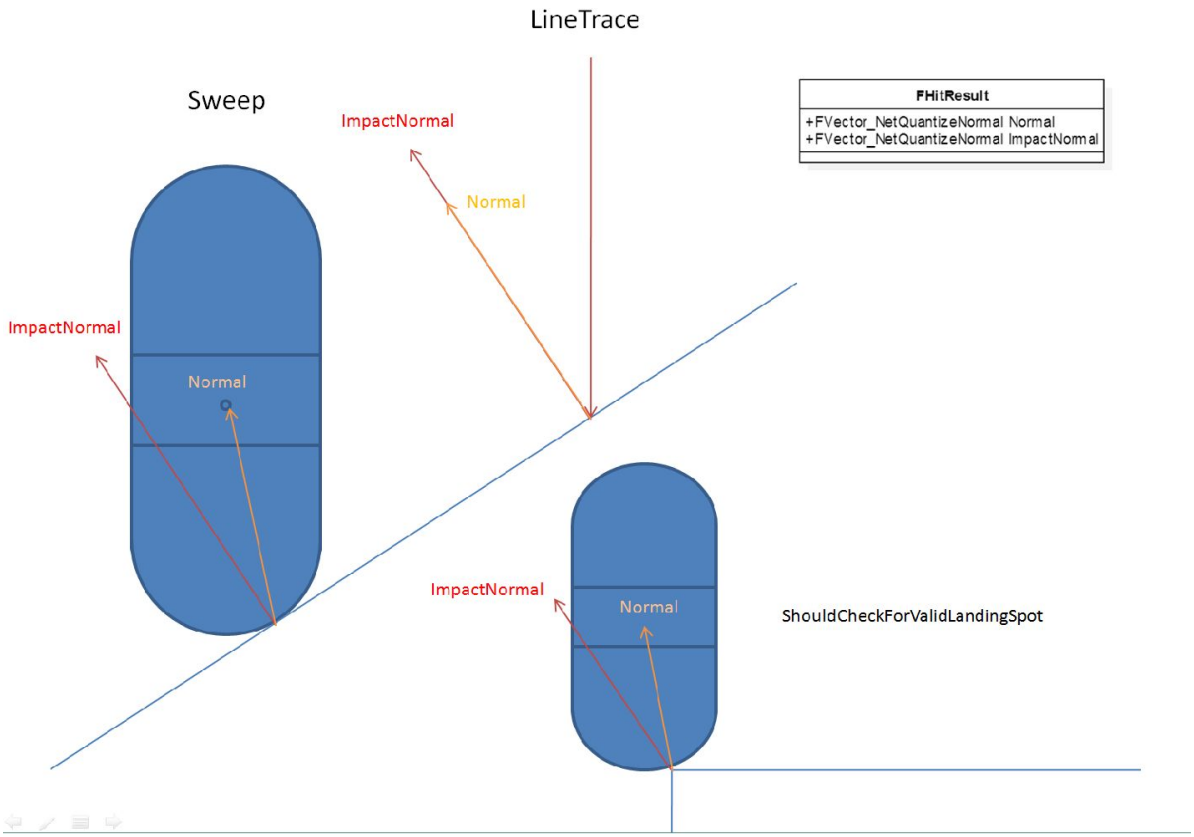


3.1 Walking

行走模式可以说是所有移动模式的基础，也是各个移动模式里面最为复杂的一个。为了模拟出出真实世界的移动效果，玩家的脚下必须要有一个可以支撑不会掉落的物理对象，就好像地面一样。在移动组件里面，这个地面通过成员变量FFindFloorResult CurrentFloor来记录。在游戏一开始的时候，移动组件就会根据配置设置默认的MovementMode，如果是Walking，就会通过FindFloor操作来找到当前的地面，CurrentFloor的初始化堆栈如下图3-2（Character Restart()的会覆盖Pawn的Restart()）：

```
UE4Editor-Engine.dll!UCharacterMovementComponent::ComputeFloorDist(const FVector & CapsuleLocation, float LineDistance, float Swee
UE4Editor-Engine.dll!UCharacterMovementComponent::FindFloor(const FVector & CapsuleLocation, FFindFloorResult & OutFloorResult, bo
UE4Editor-Engine.dll!UCharacterMovementComponent::OnMovementModeChanged(EMovementMode PreviousMovementMode, unsigned
UE4Editor-Engine.dll!UCharacterMovementComponent::SetMovementMode(EMovementMode NewMovementMode, unsigned char NewCu
UE4Editor-Engine.dll!UCharacterMovementComponent::SetDefaultMovementMode() 行 777
UE4Editor-Engine.dll!APawn::PawnClientRestart() 行 351
UE4Editor-Engine.dll!APlayerController::ClientRestart_Implementation(APawn * NewPawn) 行 713
UE4Editor-Engine.dll!APlayerController::execClientRestart(FFrame & Stack, void * const Z_Param__Result) 行 178
UE4Editor-CoreUObject.dll!UFunction::Invoke(UObject * Obj, FFrame & Stack, void * const Z_Param__Result) 行 4474
UE4Editor-CoreUObject.dll!UObject::ProcessEvent(UFunction * Function, void * Parms) 行 1308
UE4Editor-Engine.dll!AActor::ProcessEvent(UFunction * Function, void * Parameters) 行 649
UE4Editor-Engine.dll!APlayerController::ClientRestart(APawn * NewPawn) 行 4347
```

下面先分析一下FindFloor的流程，FindFloor本质上就是通过胶囊体的Sweep检测来找到脚下的地面，所以地面必须要有物理数据，而且通道类型要设置与玩家的Pawn有Block响应。这里还有一些小的细节，比如我们在寻找地面的时候，只考虑脚下位置附近的，而忽略掉腰部附近的物体；Sweep用的是胶囊体而不是射线检测，方便处理斜面移动，计算可站立半径等（参考图3-3，HitResult里面的Normal与ImpactNormal在胶囊体Sweep检测时不一定相同）。另外，目前Character的移动是基于胶囊体实现的，所以一个不带胶囊体组件的Actor是无法正常使用UCharacterMovementComponent的。



找到地面玩家就可以站立住么？不一定。这又涉及到一个新的概念PerchRadiusThreshold，我称他为可栖息范围半径，也就是可站立半径。默认这个值为0，移动组件会忽略这个可站立半径的相关计算，一旦这个值大于0.15，就会做进一步的判断看看当前的地面空间是否足够让玩家站立在上面。

前面的准备工作完成了，现在正式进入Walking的位移计算，这一段代码都是在PhysWalking里面计算的。为了表现的更为平滑流畅，UE4把一个Tick的移动分成了N段处理（每段的时间不能超过MaxSimulationTimeStep）。在处理每段时，首先把当前的位置信息，地面信息记录下来。在TickComponent的时候根据玩家的按键时长，计算出当前的加速度。随后在CalcVelocity()根据加速度计算速度，同时还会考虑地面摩擦，是否在水中等情况。

```
// apply input to acceleration
Acceleration = ScaleInputAcceleration(ConstrainInputAcceleration(InputVector));
```

五．特殊移动模式的实现思路

这一章节不是详细的实现教程，只是给大家提供常见游戏玩法的一些设计思路，如果有时间的话也会考虑做一些实现案例。如果大家有什么特别的需求，欢迎提出来，可以和大家一起商讨合理的解决方案。

5.1 二段跳，多段跳的实现

其实4.14以后的版本里面已经内置了多段跳的功能，找到Character属性JumpMaxCount，就可以自由设置了。当然这个实现的效果有点简陋，只要玩家处于Falling状态就可以进行下一次跳跃。实际上常见的多段跳都是在上升的阶段才可以执行的，那我们可以在代码里加一个条件判断当前的速度方向是不是Z轴正方向，还可以对每段跳跃的速度做不同的修改。具体如何修改，前面3.2.1小结已经很详细的描述了跳跃的处理流程，大家理解了就能比较容易的实现了。

5.2 喷气式背包的实现

喷气式背包表现上来说就是玩家可以借助背包实现一个超高的跳跃，然后可以缓慢的下落，甚至是飞起来，这几个状态是受玩家操作影响的。如果玩家不操作背包，那肯定就是自然下落了。

首先我们分析一下，现有的移动状态里有没有适合的。比如说Fly，如果玩家进入飞行状态，那么角色就不会受到重力的影响，假如我在使用喷气背包时进入Flying状态，在不使用的时候切换到Falling状态，这两种情况好像可以达到效果。不过，如果玩家处于下落中，然后缓慢下落或者几乎不下落的时候，玩家应该处于Flying还是Falling？这时候突然切换状态是不是会很僵硬？

所以，最好整个过程是一个状态，处理上也会更方便一些。那我们试试Falling如何？前面的讲解里描述了Falling的整个过程，其实就是根据重力不断的去计算Z方向的速度并修改玩家位置（NewFallVelocity函数）。重写给出一个接口MyNewFallVelocity来覆盖NewFallVelocity的计算，用一个开关控制是否使用我们的接口。这样，现在我们只需要根据上层逻辑来计算出一个合理的速度即可。可以根据玩家的输入操作（类似按键时间*燃料值*单位燃料能量）去计算喷气背包的推动力，然后将这个推动力与重力相加，再应用到MyNewFallVelocity的计算中，基本上就可以达到效果了。

当然，真正做起来其实还会复杂很多。如果是网络游戏，你要考虑到移动的同步，在客户端角色是Simulate的情况下，你需要在SimulateTick里面也处理NewFallVelocity的计算。再者，可能还要考虑玩家在水里应该怎么处理。

5.3 爬墙的实现

爬墙这个玩法在游戏里可以说是相当常见了。刺客信条，虐杀原形，各类武侠轻功甚至很多2D游戏里面也有类似的玩法。
在UE里面，由于爬墙也是一个脱离重力的表现，而且离开墙面玩家就应该进入下落状态，所以我们可以考虑借助Flying来实现。基本思路就是：

1. 创建一个新的移动模式 爬墙模式
2. 在角色执行地面移动（MoveAlongFloor）的时候，一旦遇到前面的障碍，就判断当前是否能进入爬墙状态
3. 检测条件可以有，障碍的大小，倾斜度甚至是Actor类型等等。
4. 如果满足条件，角色就进入爬墙状态，然后根据自己的规则计算加速度与速度，其他逻辑仿照Flying处理
5. 修改角色动画，让玩家看起来角色是在爬墙（这一部分涉及动画系统，内容比较多）

这样基本上可以实现我们想要的效果。不过有一个小问题就是，玩家的胶囊体方向实际还是竖直方向的，因此碰撞与动画表现可能有一点点差异。如果想表现的更好，也可以对整个角色进行旋转。

5.4 爬梯子的实现

梯子是竖直方向的，所以玩家只能在Z轴方向产生速度与移动，那么我们直接使用Walking状态来模拟是否可以呢？很可惜，如果不加修改的话，Walking里面默认只有水平方向的移动，只有遇到斜面的时候才会根据斜面角度产生Z轴方向的速度。那我这里给出一个建议，还是使用Flying。（Flying好像很万能）

玩家在开始爬一个梯子的时候，首先要把角色的Attach到梯子上面，同时播放响应的动画来配合。一旦玩家爬上了梯子，就应该进入了特殊的 **爬梯子状态**。这个状态仔细想想，其实和前面的爬墙基本上相似，不同的就是爬梯子的速度，而且玩家可以**随时停止**。

随时停止怎么做？两个思路：

1. 参考Walking移动的计算，计算速度CalcVelocity的时候使用自定义的摩擦系数Friction以及刹车速度（这两个值都设置大一些）

2. 当玩家输入结束后，也就是Acccceleration=0的时候，直接设置速度为0，不执行CalcVelocity

另外，要想让爬梯子表现的进一步好一些。看起来是一格一格的爬，就需要特殊的控制。玩家每次按下按钮的时候，角色必须完整的执行一定位移的移动（一定位移大小就是每个梯子格的长度）。这里可以考虑使用根骨骼位移RootMotion，毕竟动画驱动下比较容易控制位移，不过根骨骼位移在网络条件差的情况下表现很糟。
还有一个可以进一步优化的操作，就是使玩家的手一直贴着梯子。这个需要用IK去处理，UE商城里面有一个案例可以参考一下。