人物播放原地跳跃动画并向前移动

人物播放完动画时到达目的地

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Text : MonoBehaviour
{
    Animation anim;
    bool isJump;
    Vector3 jumpPos;
    Vector3 startPos;
    AnimationState state;
    float time;
    float deltaTime;
    void Start()
    {
        anim = GetComponent<Animation>();
        state = anim["jump"];
        state.speed = 2.0f;
        AnimationEvent event1 = new AnimationEvent();
        event1.time = state.length * 0.3f;
        event1.functionName = "JumpStart";
        state.clip.AddEvent(event1);
        anim.Play("jump");
        startPos = transform.position;
        jumpPos = transform.position + transform.forward *3f;
        time = state.length*0.3f/2;
    }

    void JumpStart()
    {
        isJump = true;
    }
    void Update()
    {
        if (isJump)
```

```
        {
            deltaTime += Time.deltaTime;
            transform.position = Vector3.Lerp(startPos, jumpPos,
deltaTime/time);
            if (deltaTime>=time)
            {
                transform.position = jumpPos;
                isJump = false;
            }
        }
    }
}

```

判定物体前方120度范围的敌人 攻击敌人击飞效果


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Text : MonoBehaviour
{
    Animation anim;
    float atkDistance=2.5f;
```

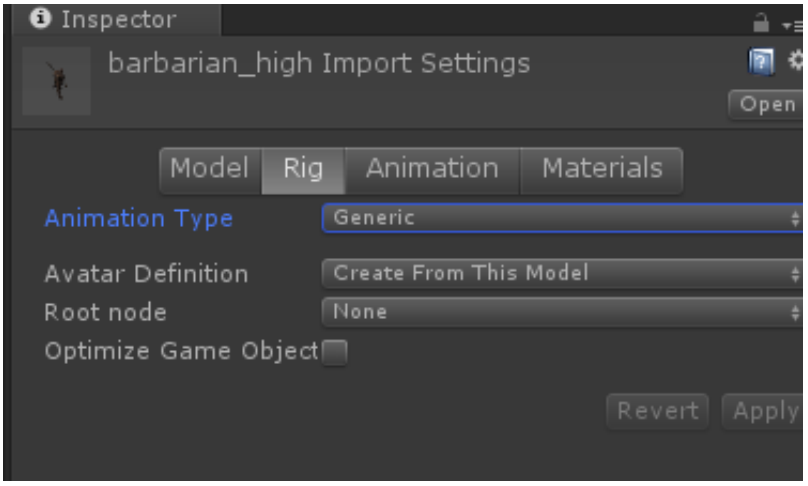
```
    float fieldOfView = 120f;

    void Start()
    {
        anim = GetComponent<Animation>();
        AnimationEvent event1 = new AnimationEvent();
        event1.time = 0.3f;
        event1.functionName = "Attack";
        anim["attack2"].clip.AddEvent(event1);
    }

    void Attack()
    {
        //相交球 以人物为中心 攻击距离为半径 构建球形碰撞器获取范围内碰撞信息
        Collider[] colliders= Physics.OverlapSphere(transform.position + Vector3.up, atkDistance, 1 << LayerMask.NameToLayer("Enemy"));

        foreach (Collider co in colliders)
        {
            //人物到敌人的向量
            Vector3 dir = co.transform.position - transform.position;
            //人物前方向量和双方相对向量所形成的角度
            float angle = Vector3.Angle(transform.forward, dir);
            if (angle <= fieldOfView / 2)
            {
                //播放被击打动画 施加力添加击飞效果
                co.GetComponent<Animation>().Play("Death");
                co.GetComponent<Rigidbody>().AddForce((dir+transform.up)* 200f);
            }
        }
    }

    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            anim.Play("attack2");
        }
    }
}
```

新版动画系统 Mecanim

1 动画控制器 Animator Controller

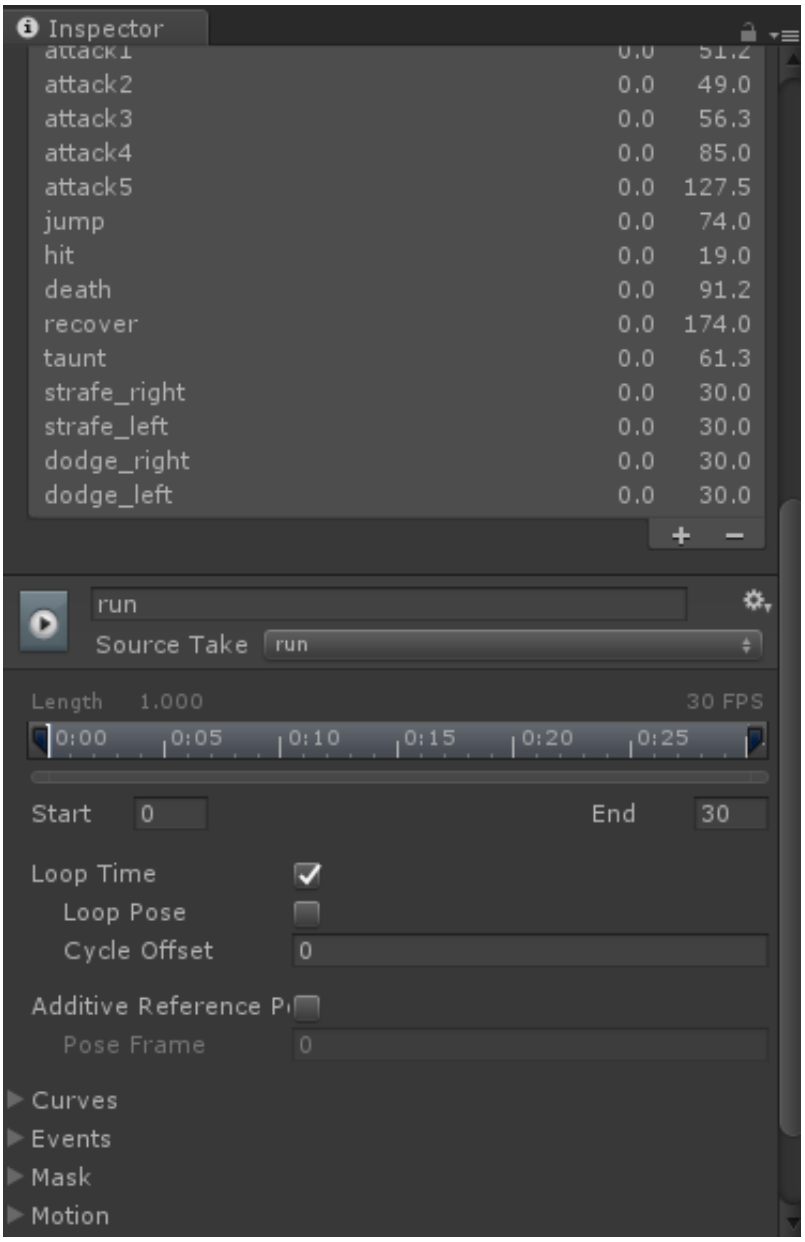
2 阿凡达 Avatar

导入的模型 FBX Rig->AnimationType->Generic(或Humanoid)

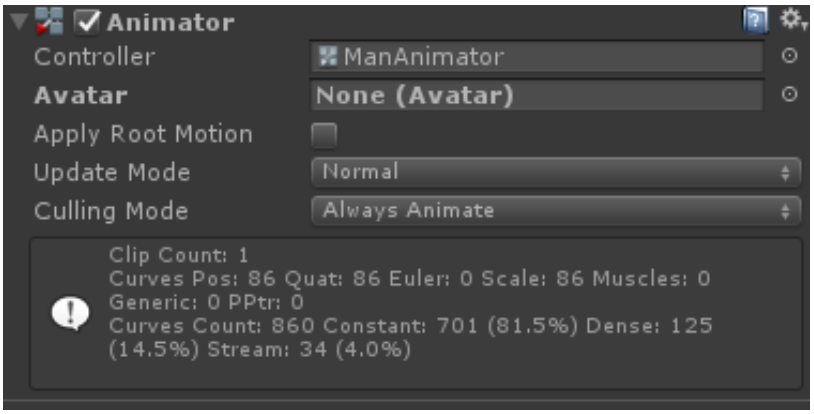


切换为新版动画以后 需要重新设置动画的循环模式

Animation->动画名->LoopTime勾选



新版动画需要使用Animator组件进行动画的切换和播放



Controller 控制器功能和Animation相同 主要控制动画的播放切换和停止

Avatar 阿凡达 如果模型本身自带动画 那么这个数据可以不用赋值

ApplyRootMotion 使用根节点运动 一般不勾选 位移大多数情况应该由程序代码控制

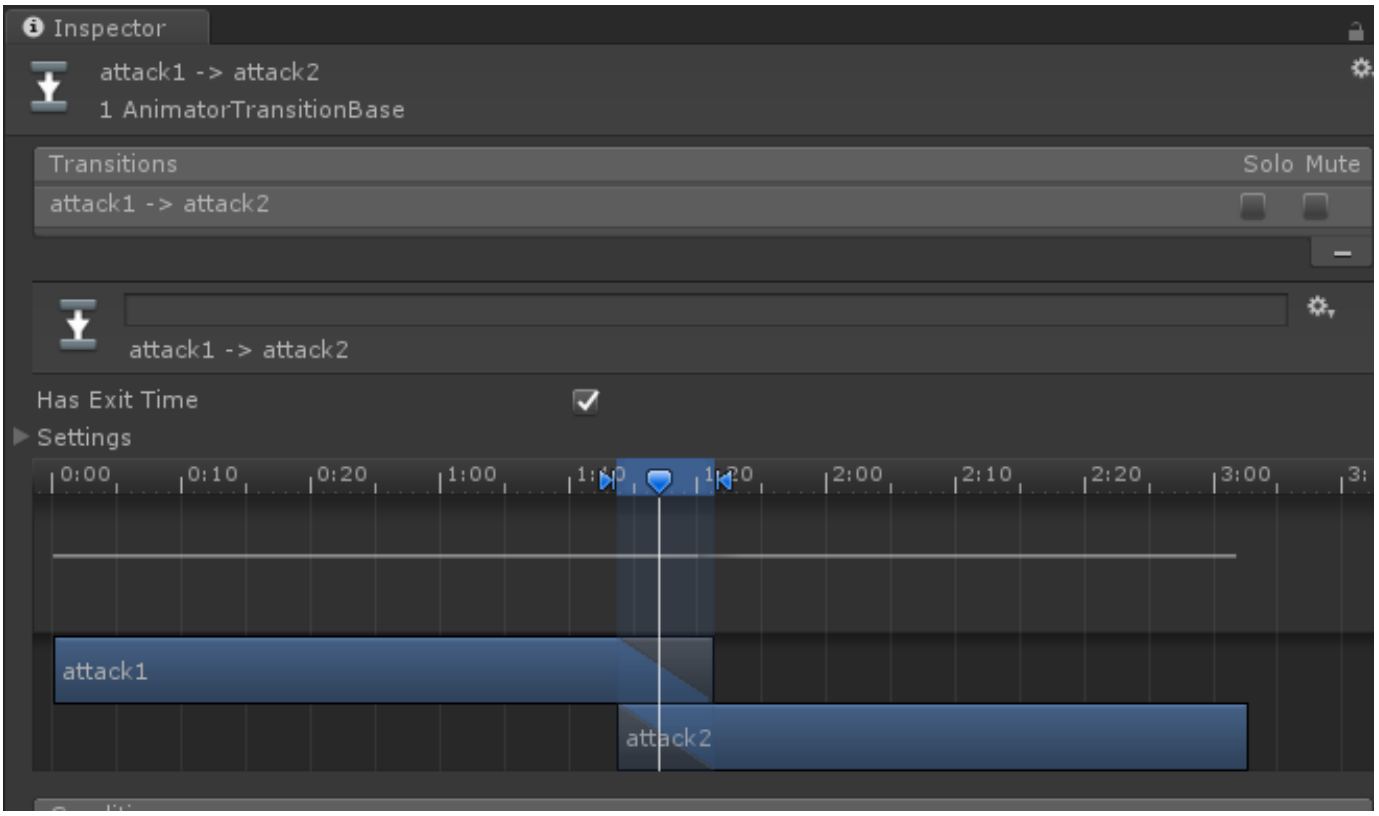
UpdateMode 默认使用Normal 动画器标准更新 独立于TimeScale之外的情况很少

设置动画的切换条件

1 设置切换值(切换时机)

2 设置动画和动画之间的过渡效果

3 HasExitTime 有退出时间的 希望前一个动画播放完以后再切换新动画 设置true



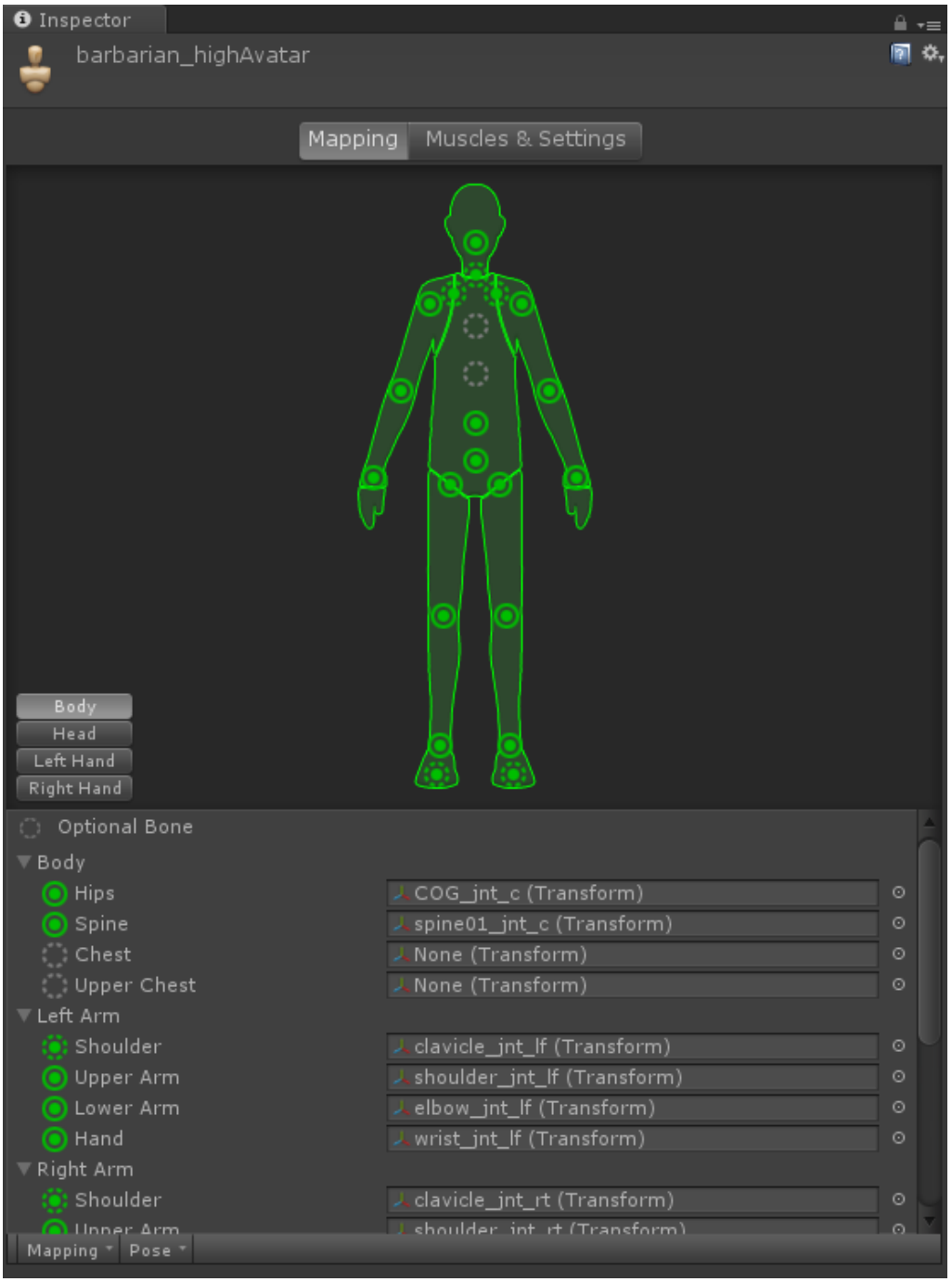

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Text : MonoBehaviour
{
    Animator animator;
    int state = 0;
    void Start()
    {
        animator = GetComponent<Animator>();
        animator.speed = 1.0f;//设置动画控制器的速度 影响所有动画控制器中需要播放的动画
    }
```

```
15
16      void Update()
17      {
18          if (Input.GetMouseButtonDown(0))
19          {
20              state++;
21              animator.SetInteger("state", state);
22          }
23
24          if (animator.GetInteger("state") == 5)
25          {
26              AnimatorStateInfo info=
    animator.GetCurrentAnimatorStateInfo(0);
27              if (info.IsName("death")&&info.normalizedTime>0.85f)
28              {
29                  state = 0;
30                  animator.SetInteger("state", state);
31              }
32          }
33      }
34  }
```

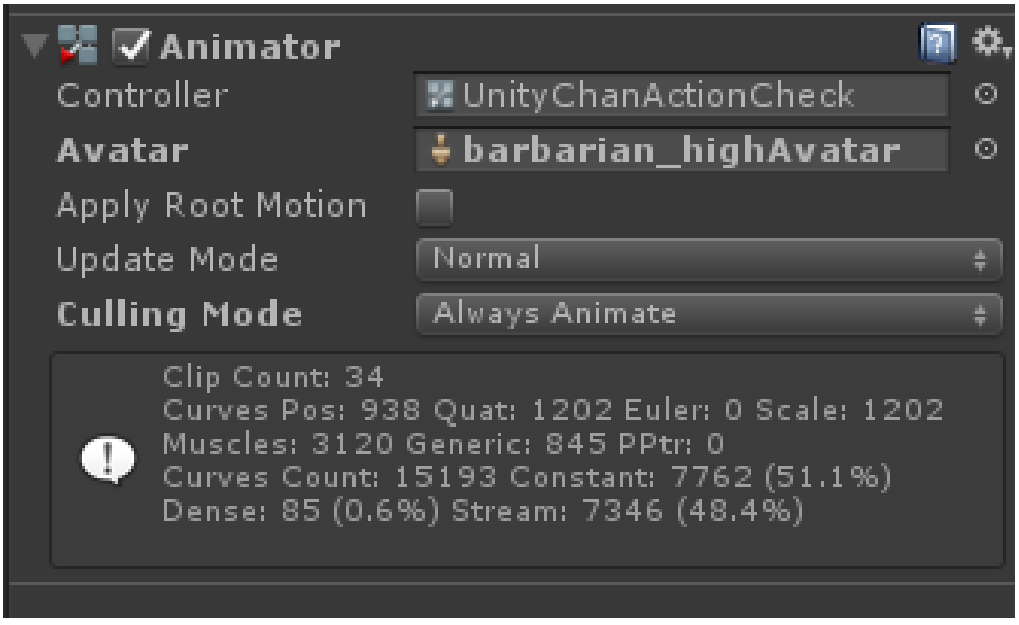
动画重用

AnimationType 设置为Humanoid 人形动画 Unity自动匹配骨骼

目的实现动画重用



匹配骨骼成功后 在Animator里设置Avatar 就可以使用其他模型的动画



FSM状态机

有限状态自动机 简单理解是一个事物的有限个状态和这些状态之间的转移和行为

以游戏举例说明：

一个怪物一般可简单分为：站立，移动，攻击，被攻击和死亡等状态

每一个状态都需要更新和改变 移动过程中每一帧的动画更新和位置改变

然后还要包含人机交互：比如监听是否有玩家对其进行了攻击或其他指令

如果发生监听到就要切换被攻击状态等的逻辑

实现状态机执行过程和判断过程是相当复杂的 要实现一个好的状态机 满足：可扩展 解耦合

传统的switch状态

```
switch(state)
case A:
    do_A();
case B:
    do_B();
case C:
    do_C();
end switch
```

利用这种状态机 实现的原理：根据当前状态执行此状态的逻辑

缺点：代码逻辑混乱 不便于维护 不够直观 耦合性太强 代码臃肿

状态机的实现方式可以参考Unity的生命周期函数

几乎每一个状态都要包含

1 进入状态 （ 播放状态动画 获取数据 ） Start

2 执行状态 （监听当前状态可能出现的事件 ，数据更新） Update

3 退出状态 (对当前状态结束时的逻辑进行处理 数据释放) Destory

状态机实现的基础：

1 多态

2 状态设计模式

3 管理者设计模式

实现人物状态机