

角色能力

本页介绍了资产中包含的各种角色能力，以及如何创建自己的能力。

- [一般信息](#)
- [标准能力](#)
- [能力概览](#)
- [组织能力](#)
- [状态机](#)
- [授权能力](#)
- [创造自己的能力](#)

一般信息

角色能力是使您的角色能够执行操作的脚本。无论是跳跃、奔跑还是按下按钮，这就是它发生的地方。拥有这些分离的能力脚本将允许最好的架构。它还将使您更轻松创建自己的能力。

标准能力

- **CharacterButtonActivation**：此组件允许您的角色与按钮供电的对象（对话区、开关.....）进行交互。这里没有什么特别的设置。
- **CharacterConeOfVision**：在角色周围投射一个锥形视野，可用于检测目标，或纯粹用于装饰目的。
- **CharacterCrouch**：允许在按下蹲伏按钮时调整控制器（和专用动画）的大小
- **CharacterDash2D/3D**：此能力允许角色向指定方向冲刺。您可以决定冲刺模式、指定用于移动角色的曲线动画、冲刺的持续时间、应覆盖的距离等。
- **CharacterDamageDash2D/3D**：此能力类似于常规的冲刺能力，但也会在冲刺时造成伤害。为此，此功能启用/禁用应该创建、放置在角色下方并绑定到 DamageDash 的检查器中的 TargetDamageOnTouch 插槽的 DamageOnTouch 对象。如果您正在查看有关如何扩展现有能力以向其添加更多功能的示例，则此能力也是一个很好的参考。
- **CharacterDirectionMarker**：此功能可让您将角色上的对象朝向瞄准方向或移动

方向，以及在上校演示角色上的两个示例

- **CharacterFallDownHoles2D** : 一个只有 2D 的能力，它会让你的角色掉下"洞"，由你在 TopDownController2D 的检查器中指定的洞图层蒙版定义。
- **CharacterGridMovement** : 让您在网格上行走（这意味着您的角色将始终完全以网格单元为中心停止移动）。这适用于 2D 和 3D，并且需要在场景中存在并正确设置 GridManager。您可以在 Minimal2DGrid、Minimal3DGrid 和 Explodudes 演示场景中找到该设置的示例。请注意，这是与"常规" CharacterMovement 不同的系统，并且大多数与移动相关的能力（跳跃、冲刺）或 AI 动作不适用于网格移动。
- **CharacterHandleWeapon** : 让你的角色装备和使用武器，无论它是否来自库存。
- **CharacterHandleSecondaryWeapon** : 同样的事情，但多了一件武器。
- **CharacterInventory** : 让您的角色将自己绑定到库存，以便能够装备武器等等。请注意，该引擎目前不支持多人游戏库存，但已被大量要求并即将进行更新。
- **CharacterJump2D/3D** : 当你按下跳跃按钮时会让你的角色跳跃。请注意，3D 版本实际上会移动您角色的控制器，而 2D 版本会将其保持在原位，动画负责跳跃错觉。请注意，在这两种情况下，您的角色在跳跃时不再被视为接地。
- **CharacterMovement** : 基本的、基于地面的运动。您将能够从其检查器中指定步行速度、空闲阈值、加速和减速、足迹粒子等。您还可以强制自由、2、4 或 8 个方向移动。
- **CharacterOrientation2D/3D** : 将旋转或翻转您的角色，使其面向运动方向、武器方向或两者。
- **CharacterPathfinder3D** : 一个只有 3D 的能力，会让你的角色在导航网格上找到一条路径。不过，该导航网格需要先出现在场景中，然后才能使用。
- **CharacterPathfindToMouse** : 仅限 3D，让您点击地面并让您的角色移动到该目标，以及在 LoftSuspendersMouseDriven 演示角色上的演示
- **CharacterPause** : 允许具有此能力的角色（以及控制它的玩家）使用暂停按钮暂停游戏
- **CharacterPersistence** : 让您在转换到新场景时让角色保持其确切的当前状态
- **CharacterRotateCamera** : 此功能可让您围绕角色在垂直轴（2D 中的 z，3D 中的 y）上旋转相机。它还提供旋转输入以匹配相机方向、确定旋转空间和速度的选项，以及武器的专用瞄准选项。您将在 Loft3D 演示场景中找到它的示例（使用 L 和 M 旋转相机）。
- **CharacterRotation2D** : 让您的 2D 角色更改其模型的旋转以匹配其前进的方向。
- **CharacterRun** : 让你的角色在按下运行按钮时以指定的速度奔跑
- **CharacterSwap** : 此功能将允许您在单个场景中交换对多个角色的控制。有关示例，请参阅 Minimal2DCharacterSwap 演示场景。请注意，此功能取决于 LevelManager 的正常角色实例化。
- **CharacterSwitchModel** : 允许您为另一个模型切换角色的外观。

- **CharacterTimeControl**: 让您的角色在按下"时间控制"按钮时将当前时间刻度更改为检查器中指定的时间刻度，在您选择的持续时间内，无论是否进行。
- **角色能力节点交换**: 此能力可让您指定一组新能力，只需按一下按钮即可进行交换。

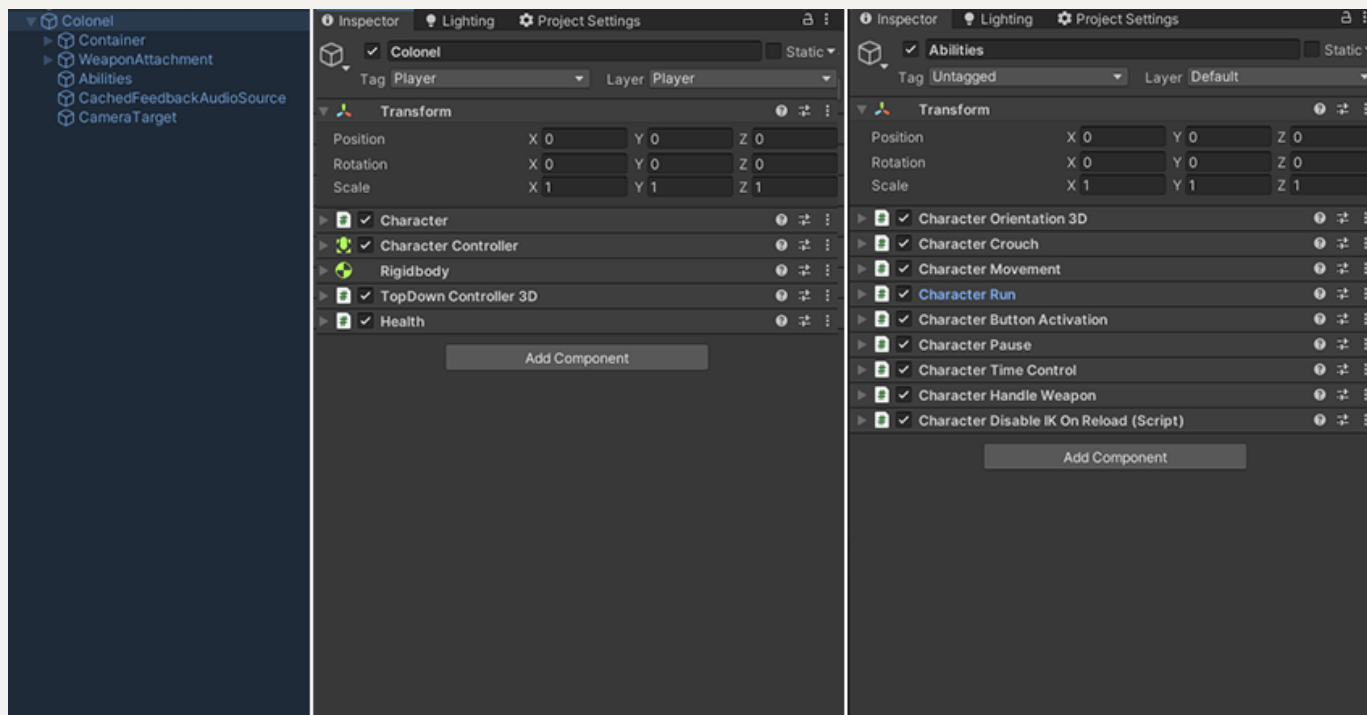
能力概览

那么能力有什么作用呢？如果您对代码的工作方式感兴趣，最简单的找出方法是查看 `CharacterAbility.cs` 类，所有能力都从该类继承。它有几个方法，让我们快速浏览一下主要方法：

- **初始化**: 顾名思义，这是基类将获得角色或场景的一部分，这些部分将定期用于儿童能力。像相机、`TopDownController` 组件、输入管理等东西。子能力通常会使用它来获取附加组件或初始化变量（跳跃次数等）。
- **动画方法**: `InitializeAnimatorParameters` 和 `UpdateAnimator` : 使用第一个注册动画参数，第二个更新它们。这是分两步完成的，以避免每帧检查每个参数是否存在，这最终会导致性能问题。
- **HandleInput**: 被每个检查相关按钮和轴状态的能力覆盖。如果按下/释放某个按钮，此方法将调用该能力中的其他方法。
- **Early/Process/Late Process 能力**: 这些方法在每次更新时由状态机调用。
- **重置**: 当角色死亡时将调用此方法。用于重置计数器等。
- **Play/Stop Sfx**: 用于触发能力声音的方法。默认情况下，每个能力都有 3 种声音（在其检查器中按能力定义）：一种是在开始时，一种是在使用时，一种是在停止时。您当然可以只使用其中一种或不使用其中一种。如果您创建自己的能力，则需要调用这些方法来触发声音。

组织能力

从 v1.10 开始，您现在可以在多个游戏对象（或节点）之间拆分能力。



上校预制件及其分裂能力节点

设置起来很容易。默认情况下，角色组件将在其自己的节点（通常是角色层次结构的顶层）上寻找能力。此外，您可以将任意数量的附加节点绑定到它的 **AdditionalAbilityNodes** 数组。通常您会有（如上图所示）许多带有能力的空游戏对象。只需将它们拖到该数组中，Character 组件就会在初始化时注册它们。

这对很多事情都有用。它避免了在单个对象上有太多组件，这在寻找特定能力或领域时会变得混乱。它还允许您一次启用/禁用整个能力节点，甚至交换整个能力集。您将在上校演示场景中找到上校预制件就是这样构建的。

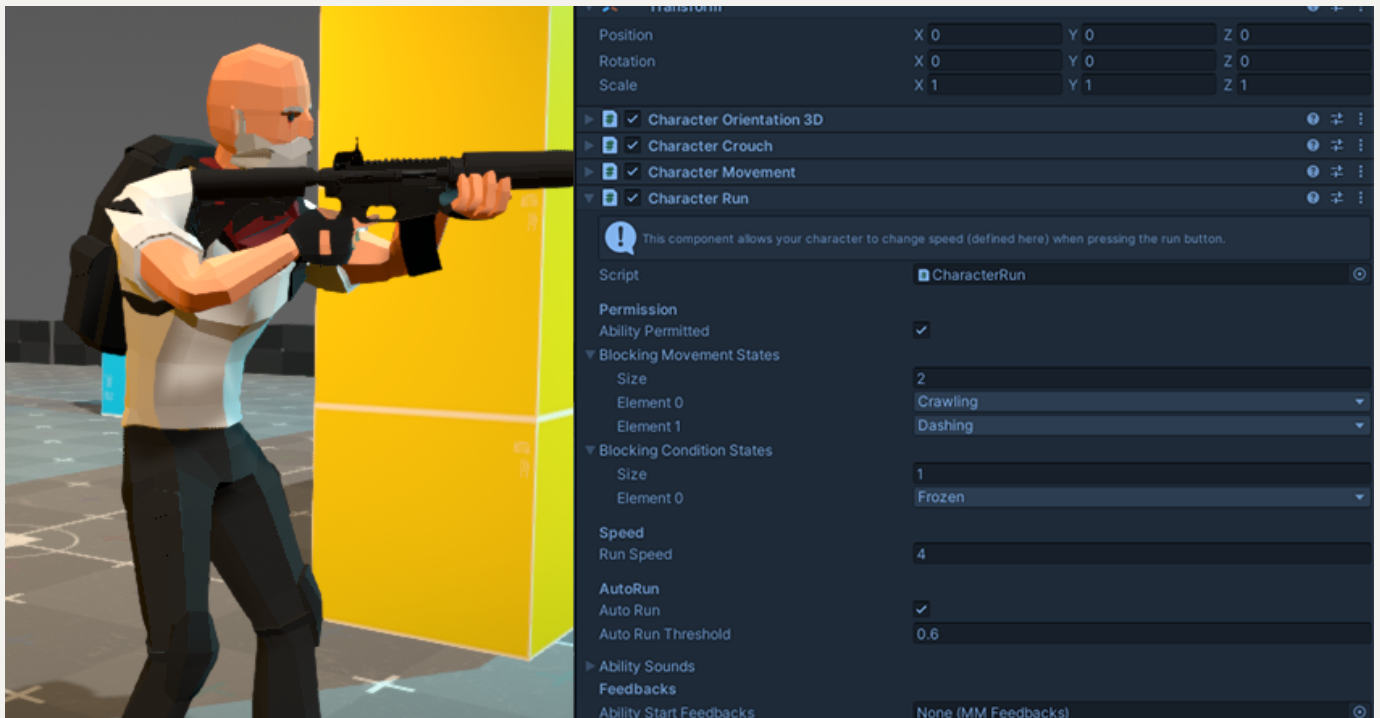
状态机

Character 组件负责触发各种能力。它使用状态机来实现。StateMachine 是一种设计模式，它基本上会存储当前状态和前一个状态（如果您想了解更多信息，请查看代码或 API 文档）。默认情况下，角色使用两个状态机：

- **MovementState**：通过任何技能的 `_movement` 属性访问，它代表角色正在执行的当前动作。
- **ConditionState**：通过任何技能的 `_condition` 属性访问，它存储角色的当前状态（正常、死亡、暂停等）。

它的工作方式是在场景开始时，角色将初始化所有能力，然后每一帧，调用它们的 EarlyProcess、Process 和 LateProcess 方法，并最终在死亡时重置它们。其他状态机实现通常只会在更新时调用当前状态的能力。出于多种原因，这个没有，但主要是为了便于扩展系统，而不必重写所有内容或修改现有类。这意味着每个能力都负责处理自己的输入，防止进入其方法（通过测试当前状态是否允许 - 例如，您不能在未接地的情况下行走）。引擎中包含的大多数功能不使用 EarlyProcess 或 LateProcess，但如果您需要，它仍然是一种可能性。

授权能力



对上校预制件的运行能力添加额外限制

大多数能力定义了它们自己的基本限制，这些限制是上述状态机中的状态，您无法从中过渡到该能力。例如，默认情况下，您不能在喷气背包时跳跃。这些默认限制涵盖了最需要的用例，但在您的游戏中，您可能希望进一步限制。这可以通过扩展更改条件检查的基本能力来完成，或者简单地从检查器中通过将状态添加到阻止移动状态和阻止条件状态列表来完成，如上图所示。

创造自己的能力

创建自己的能力的最简单方法是扩展 **CharacterAbility**，就像现在引擎中的所有能力一样。在需要时覆盖方法（确保在开始时调用基本方法）。要测试您的新能力，您只需将其添加到现有角色中，它就会自动添加到状态机中，并像其他角色一样进行处理。要记住的一件事是与其他能力的相互作用。您可能希望扩展其他功能以防止或授权某些状态更改。此外，您的能力可能需要新的状态。您可以在 `CharacterStates.cs` 中声明这些（在 `MovementStates` 或 `CharacterConditions` 枚举中的任何位置，顺序无关紧要）。

要获得灵感，您可以查看当前的角色能力，因为它们正是您要创建的内容，并且是您可以实现的目标的好例子。这是我用来创建它们的一个很好的基础，因为我发现这些方法是我用来覆盖最多的方法。确保你用你的东西替换了所有的 TODO：

```
using UnityEngine;
using System.Collections;
using MoreMountains.Tools;

namespace MoreMountains.TopDownEngine // you might want to use your
own namespace here
{
    /// <summary>
    /// TODO_DESCRIPTION
    /// </summary>
    [AddComponentMenu("TopDown
Engine/Character/Abilities/TOD_REPLACE_WITH_ABILITY_NAME")]
    public class TOD_NEW_ABILITY_NAME : CharacterAbility
    {
        /// This method is only used to display a helpbox text
        /// at the beginning of the ability's inspector
        public override string HelpBoxText() { return
"TODO_HELPBOX_TEXT."; }

        [Header("TODO_HEADER")]
        /// declare your parameters here
        public float randomParameter = 4f;
        public bool randomBool;

        protected const string _yourAbilityAnimationParameterName =
"YourAnimationParameterName";
```

```

protected int _yourAbilityAnimationParameter;

/// <summary>
/// Here you should initialize our parameters
/// </summary>
protected override void Initialization()
{
    base.Initialization();
    randomBool = false;
}

/// <summary>
/// Every frame, we check if we're crouched and if we still
should be
/// </summary>
public override void ProcessAbility()
{
    base.ProcessAbility();
}

/// <summary>
/// Called at the start of the ability's cycle, this is where
you'll check for input
/// </summary>
protected override void HandleInput()
{
    // here as an example we check if we're pressing down
    // on our main stick/direction pad/keyboard
    if (_inputManager.PrimaryMovement.y < -
_inputManager.Threshold.y)
    {
        DoSomething();
    }
}

/// <summary>
/// If we're pressing down, we check for a few conditions to
see if we can perform our action
/// </summary>

```

```

protected virtual void DoSomething()
{
    // if the ability is not permitted
    if (!AbilityPermitted
        // or if we're not in our normal stance
        || (_condition.CurrentState !=
CharacterStates.CharacterConditions.Normal)
        // or if we're grounded
        || (!_controller.Grounded))
    {
        // we do nothing and exit
        return;
    }

    // if we're still here, we display a text log in the
console
    MMDebug.DebugLogTime("We're doing something yay!");
}

/// <summary>
/// Adds required animator parameters to the animator
parameters list if they exist
/// </summary>
protected override void InitializeAnimatorParameters()
{
    RegisterAnimatorParameter(_yourAbilityAnimationParameterName,
AnimatorControllerParameterType.Bool, out
_yourAbilityAnimationParameter);
}

/// <summary>
/// At the end of the ability's cycle,
/// we send our current crouching and crawling states to the
animator
/// </summary>
public override void UpdateAnimator()
{

```



```
        bool myCondition = true;
        MMAnimatorExtensions.UpdateAnimatorBool(_animator,
        _yourAbilityAnimationParameter, myCondition,
        _character._animatorParameters);
    }
}
```