# Mario Forever

# Thunder Editor Refactored

# 使用手册（中文版）

# 阅读前必读

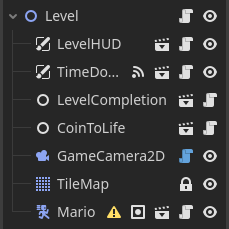
本手册旨在向开发者介绍本引擎中一些物件的结构与实现方法，诸如Godot的基本操作、代码编写等，请各位开发者自行前往[官方文档](https://docs.godotengine.org/zh-cn/4.x/)进行学习。

# 关卡结构

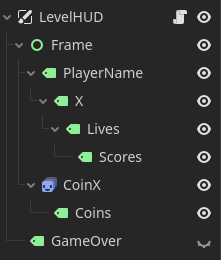
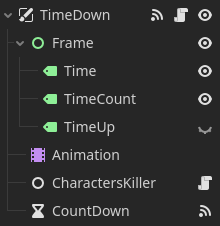
## 1.1 关卡结构初览

关卡的节点排布可以说是引擎最核心的部分，了解关卡制作结构便十分必要。

下图为关卡的节点排布结构图：



接下来我们从上往下依次详细介绍一下每个部件：

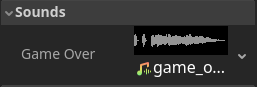
* **LevelHUD**：即关卡中用于显示文字信息的物件，理论上可以删除。其内部结构如下图所示：  
    
  其中Frame用于将元件位置与窗口大小相适配，比如，你将窗口宽度调整为864，那么所有靠近窗口右侧的物件都会进行位移，使其坐标依旧保持与窗口右侧保持一定距离。  
  下图为该物件的编辑器显示：  
  
* **TimeDown**：用于显示关卡剩余时间的物件，**如果关卡不需要限时，请删除此物件**。其结构如下图所示：  
    
  其中Frame用于将元件位置与窗口大小相适配，Animation负责控制Time节点的纵向缩放动画（即倒计时不足100点时所播放的动画），CharactersKiller负责在倒计时为0时强制杀死所有玩家。CountDown用于倒计时，其timeout信号连接在TimeDown上，用于倒计时，其默认计时间隔为0.5s。  
  该物件的编辑器显示如下图所示：  
    
  注意：如果倒计时结束后不需要杀死所有角色，请将TimeDown的time\_over信号与CharactersKiller的连接断开。同时，由于**此操作会导致其所有实例均会产生相同修改**，因此建议创建副本后使用副本进行该操作并实例化至关卡场景内。
* **LevelCompletion**：负责关卡通关的组件，**每个关卡必须存在，不得删除！**
* **CoinToLife**：负责将金币转化为命数的组件，默认100金币转化为1条命，**不建议删除**。
* **GameCamera2D**：负责关卡滚动的组件，可以一次存在多个（详细用法会在后文介绍），但关卡中**至少要有一个该组件**。（如果关卡为强制卷轴关卡，也可以替换为**AutoScrollCamera2D**组件）
* **TileMap**：绘制瓦片贴图的核心组件。有了TileMap，你就可以直接绘制大片地形，而不需要像CTF那样通过复制粘贴来制作地形。
* **Mario**：不解释了，玩家。

## 1.2 核心组件详解

接下来将较为详细地介绍一下上述组件的运作机理：

### 1.2.1 LevelHUD

其实这个LevelHUD本身没有多少参数，只不过开发者可以在检查器当中调整其播放音效：



接下来我们看一下该物件在就绪初始化时的代码：

1. Character.Data.get\_signals().data\_updated.connect(func(type: Character.Data.DataSignal.Value, value: Variant) -> void:
2. match type:
3. Character.Data.DataSignal.Value.LIVES:
4. const DEFAULT: StringName = &"PLAYER"
6. @warning\_ignore("incompatible\_ternary")
7. player\_name.text = (DEFAULT if Character.Getter.get\_characters(get\_tree()).is\_empty() else Character.Getter.get\_character(get\_tree(), 0).nickname.to\_upper()).left(8)
9. lives.text = str(value)
10. Character.Data.DataSignal.Value.SCORES:
11. scores.text = str(value)
12. Character.Data.DataSignal.Value.COINS:
13. coins.text = str(value)
14. , CONNECT\_DEFERRED)
16. Events.EventCharacter.get\_signals().game\_over.connect(func() -> void:
17. game\_over.visible = true
19. await Sound.play\_1d(sound\_game\_over, self).finished
20. await get\_tree().create\_timer(1, false).timeout
22. # TODO: After-game-over executions
23. )
25. Character.Data.init\_data()

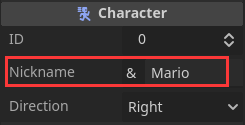
上述代码中，Character.Data.get\_signals()用于获取玩家数据库的信号库，其中有一个叫data\_updated的信号，在玩家数据发生更新时发送，其带有两个参数：一个叫type，告诉接收方这个数据是什么；另一个叫value，即该数据所对应的值是多少。下列代码为type这个参数的类型定义：

1. enum Value {
2. LIVES,
3. SCORES,
4. COINS,
5. TOTAL\_COINS,
6. DEATH\_COUNTS
7. }

可以看到，type这个参数的类型为枚举，其中的枚举常量便顾名思义对应了数据的类别。比如，如果type这个参数为Value.LIVES，那么就表示玩家的命数发生了更新，这个时候系统就会发送与玩家命数有关的data\_updated信号。

之后使用match分支检测输入的类型，减少代码重复量。

在处理命数时，对玩家名称也有所顾及，玩家名称可在Mario物件中进行修改：



第7行代码中，默认显示id为0的玩家（下称**队长**）的名称，且默认最多不超过8个字符，否则会影响HUD的显示效果。同时，如果当前关卡无任何玩家，则会显示默认名称”PLAYER”。

为了防止意外情况出现，特地使用了CONNECT\_DEFERRED来延迟调用信号所连接的匿名方法。

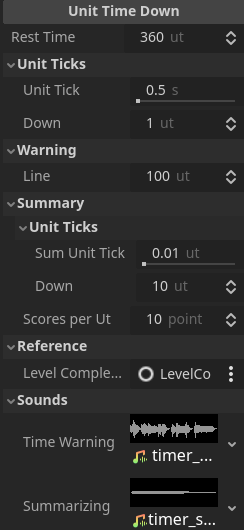
第16行开始为处理Game Over事件的代码，这里获取了全局类Events.EventCharacter的game\_over信号，并将其连接在一个匿名方法上，其中Sound负责音效播放，调用其play\_\*d()方法之后会返回一个AudioStreamPlayer(2D/3D)实例，此时使用await等待该实例将音效播放完毕，之后使用await get\_tree().create\_timer(1, false).timeout等待1s后决定是否重启游戏等操作（待实装）。

最后，在所有信号连接全部就绪后，就需要调用Character.Data.init\_data()方法来强制系统发送data\_updated信号，在游戏开始时更新HUD的显示信息。

## 1.2.2 TimeDown

这个节点讲起来比较复杂，因为涉及到十分复杂的底层些代码，这里我们只介绍其参数、信号以及其基本作用原理。

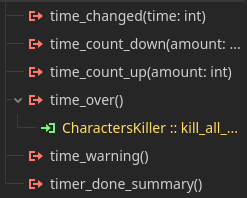
首先是其参数，如图所示：



看起来会比较复杂：

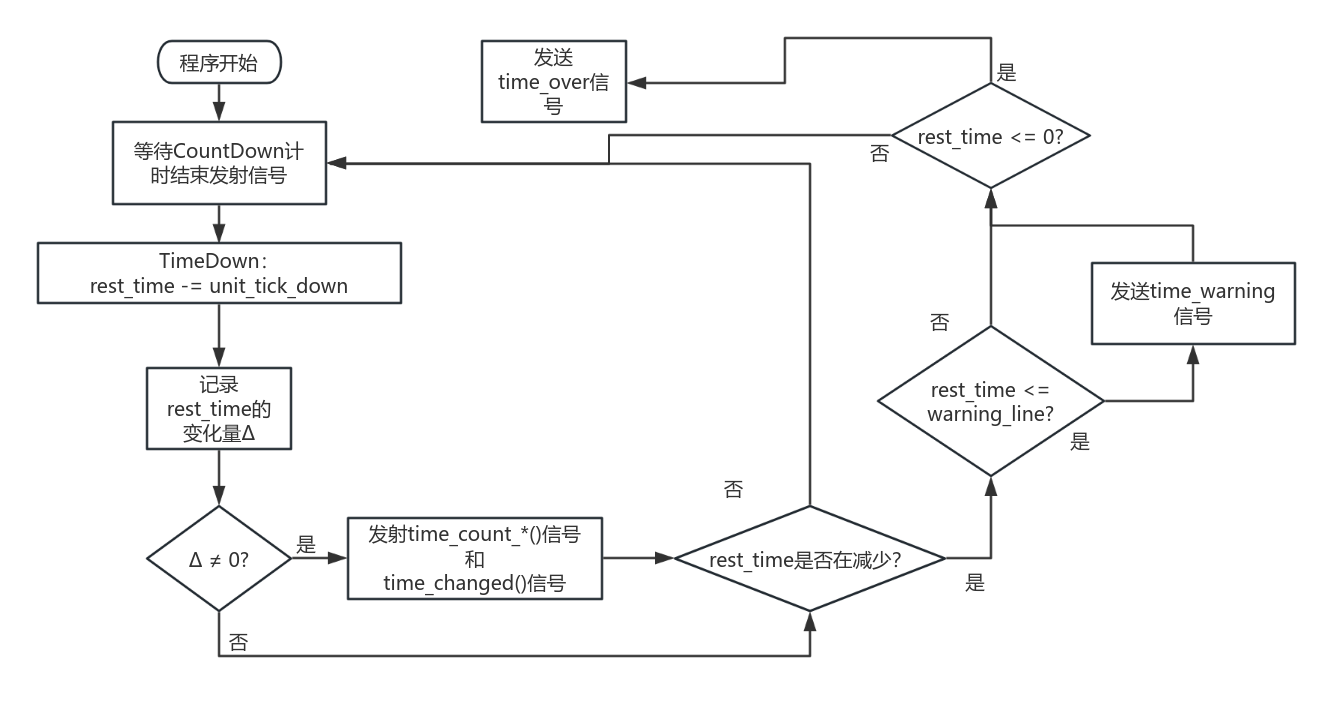
* rest\_time：即剩余时间，单位为ut（见下方unit\_tick）。当时间为0时发射time\_over信号。
* unit\_tick：单位减少时间，也即时间单位，默认每0.5s减少1ut（即默认1ut = 0.5s）。
* unit\_tick\_down：单次减少的单位时间，即默认每0.5s减少多少ut的时间，默认为减少1ut。
* warning\_line：当rest\_time在下降过程中小于等于这个数值时便会触发警告动画，同时发射time\_warning信号。
* sum\_unit\_tick：在进行剩余时间结算时的单位减少时间。
* sum\_unit\_tick\_down：在进行剩余时间结算时单次所减少的单位时间。
* sum\_scores\_per\_ut：**每减少一点单位时间**能增加的分数。也就是说：如果sum\_unit\_tick\_down为10ut，则实际上会结算为100分而非10分。
* level\_completion\_path：指向LevelCompletion组件的节点路径，**建议不要更改，并保持该组件节点与其位于同一节点下**。
* 最后两个分别是在触发警告动画时所播放的音效，以及在进行剩余时间结算时所发出的音效。

接下来是该组件的信号：



* time\_changed(time: int)：在rest\_time发生更改时发出，time为发生变更后的时间。
* time\_count\_down(amount: int)：在rest\_time减少时发出，amount为减少量。
* time\_count\_up(amount: int)：在rest\_time增加时发出，amount为增加量。
* time\_over：在rest\_time减少到0时发出。
* time\_warning：在rest\_time减少至小于等于warning\_line时发出。
* time\_done\_summary：在剩余时间结算完成后发出。

由于TimeDown组件的内部代码过于复杂，因此此处不再对代码架构进行说明。下列流程图便代以为开发者介绍该组件的运作机理：



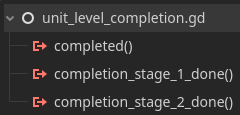
## 1.2.3 LevelCompletion

前面提到过，该组件是关卡的核心组件，负责处理通关，其属性如下图所示：



* completion\_delay：当触发通关后（一般为播放通关音效时开始）延迟多少秒后执行剩余时间结算的等操作（如果关卡中有TimeDown组件）。
* character\_direction：触发通关后，角色的朝向。
* character\_walking\_speed：触发通关后，角色的行走速度，为正值。
* next\_scene：通关处理结束后所要进入的的场景tscn文件路径，文件名必须以”room\_”开头。
* 以及通关时所播放的音效。

该组件的信号如下图所示：



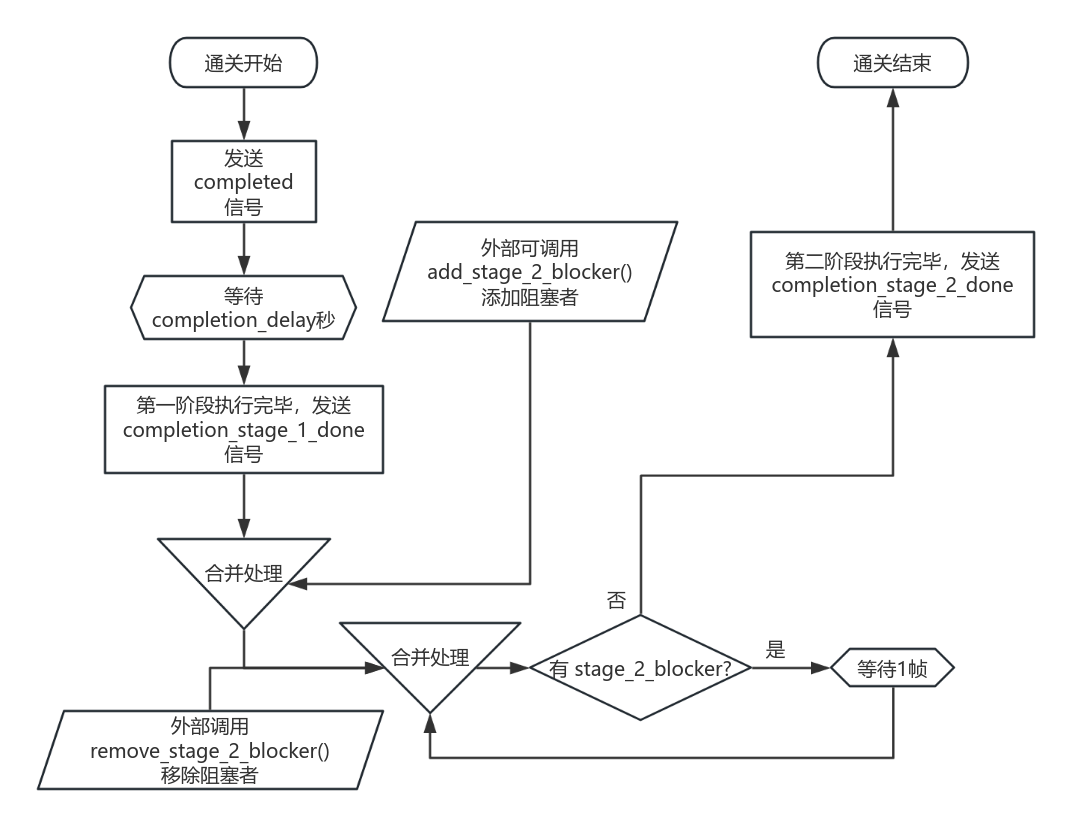
* completed：在触发通关时立即发出。
* completion\_stage\_1\_done：在completion\_delay的延迟执行完毕时发出。（第一阶段）
* completion\_stage\_2\_done：一般为在剩余时间结算结束后发出。（第二阶段）

你可能已经注意到了，在关卡结算的时候有两个阶段：

第一阶段：此时关卡一般会播放通关音效，在completion\_delay所规定的延迟执行完毕后，进入第二阶段。

第二阶段：此时关卡一般在进行结算操作，待结算完成后视为通关。

该作用机理的流程图如下所示：

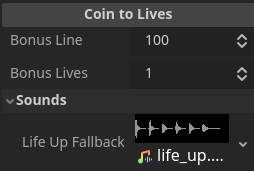


注意：上图中添加/移除阻塞者的操作可以在任意时刻进行，且并非多线程操作。其中的stage\_2\_blocker便为所谓的“阻塞者”。

TimeDown进行结算，本质上就是将LevelCompletion的completed信号连接在了某个方法上，让TimeDown的CountDown子节点暂停计时，然后将completion\_stage\_1\_done连接在某个方法上，使其进入结算状态，在进入结算状态之前，TimeDown就调用了LevelCompletion的add\_stage\_2\_blocker()方法（参数为self）以阻塞其立即结束第二阶段，等TimeDown结算完毕后，调用LevelCompletion的remove\_stage\_2\_blocker()方法（参数为self），取消阻塞。在阻塞期间，LevelCompletion实际上是进入了一个小的帧循环状态，每一帧都会检查是否还存有阻塞者，在结算结束后，阻塞者被清空，此时就会结束第二阶段。

## 1.2.4 CoinToLife

这个组件也是相对比较简单的一个。其属性如下图所示：



* bonus\_line：当金币数超过该数值时会奖命。
* bonus\_lives：当金币数超过该数值时会奖多少条命，默认1条命。
* 以及一个奖命音效。

其代码实现如下所示（位于\_ready()函数内）：

1. Character.Data.get\_signals().data\_updated.connect(func(type: Character.Data.DataSignal.Value, value: int) -> void:
2. if type == Character.Data.DataSignal.Value.COINS:
3. var coins := value
4. var given\_lives := 0
6. if coins >= bonus\_line:
7. while coins >= bonus\_line:
8. coins -= bonus\_line
9. given\_lives += bonus\_lives
11. Character.Data.coins = coins
13. var character := Character.Getter.get\_random(get\_tree(), 1)[0]
14. if is\_instance\_valid(character):
15. var lives := \_Lives.instantiate()
16. lives.amount = given\_lives
17. lives.global\_position = character.global\_position
18. character.add\_sibling.call\_deferred(lives)
19. else:
20. Character.Data.lives += given\_lives
22. Sound.play\_1d(sound\_life\_up\_fallback, self)
23. )

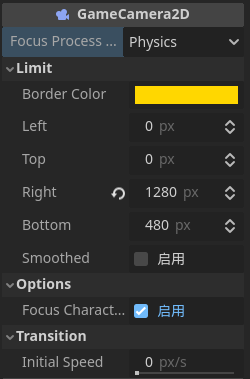
这段代码类似前面1.2.1章节中的代码。第6行负责处理超出bonus\_line的金币数以及奖命，由于可能会因为一些特殊情况导致命数大于等200，此时并不会让金币小于bonus\_line，因此需要一个while循环来处理超出的部分，并确保最终数值小于bonus\_line。Thunder Editor Refactored在处理显示奖命时，如果当前场景中含多名角色，则会随机抽取一个角色进行显示。

需要注意的是：Thunder Editor Refactored的金币数据分为**有效金币数**（COINS）和**金币总数**（TOTAL\_COINS）两种，前者用于奖命，而后者则可用于制作商店等售卖系统时使用。

## 1.2.5 GameCamera2D

相机是平台游戏中必不可少的元件之一，在Thunder Editor Refactored中，建议初学者使用GameCamera2D或者AutoScrollCamera2D来制作相机。

下图为GameCamera2D相机的属性：



* focus\_process\_mode：相机跟踪时所采用的帧循环类型，默认为physics，因为玩家的位移等处理是在physics\_process中进行的，因此设成physics也便于与玩家同步变换信息，从而准确地进行跟踪。
* *limit\_border\_color*：用于在编辑器中显示滚屏范围矩形区域的显示色。
* limit\_\*：滚屏边界坐标，**可正可负**。
* focus\_characters：启用后，相机将自动跟踪玩家。（**AutoScrollCamera2D默认将该属性关闭**）。
* transition\_initial\_speed：若该值大于0，则表示启用平滑过渡。在角色进入滚屏区域后，系统会自动切换当前滚屏区域，在此期间会有一段插值过渡，便为前文所述的平滑过渡。单位为px/s。

同一场景内可以存在多个相机，但**有且只能有一个为激活状态**。你可以在当前关卡中创建该组件的副本，然后调整其滚屏区域边界坐标即可完成多滚屏边界的创建。