

碎片

场景——用于存放节点与运行节点的数据类型



用来运行主场景



仅运行当前正在编辑的场景



给选中节点附加脚本



对节点可进行的操作

在视图区上方有个

文的意思就是

2d视图

3d视图

脚本视图

资产库视图

Godot的script语言

GDS负责实现内在逻辑，同时控制节点展示具体的功能

是GD自己的语言

GDS的编写

自定义变量：var是关键字

变量命名规则和py差不多

定义函数：func是关键字

有一些固定的函数名，比如_enter_tree()

这是用来节点进入场景树的时候自动被触发与运行其中的语句

```

extends Node
func _enter_tree():
    >| print(114514)
    >| pass
    
```



比如然后点然后把debug窗口关闭就可以看到了

```

Godot Engine v4.3.stable.official.77dcf97d8 - https://godotengine.org
Vulkan 1.3.280 - Forward+ - Using Device #0: NVIDIA - NVIDIA GeForce RTX 4060 Laptop GPU

114514
--- Debugging process stopped ---
    
```

命名同上

当然也可以进行变量运算

```

extends Node
var kk = 1
func _enter_tree():
    >| kk += 1
    >| print(114514+kk)
    >| pass

```

有内置函数比如print

即使是同一脚本文件，绑定不同的对象从而产生了同名的变量，这两个变量以及变量代表的数
据也分别属于不同的对象，互相独立。因此使用变量前必须指明变量所属的对象。

```

self.kk += 1
print(114514+self.kk)

```

这个self表示“自己”的变量

但程序可以允许省略这个self.，但是需要在使用前指明哪个对象的变量

- 数据类型
 - 整型int
 - 浮点数float
 - 字符串string
 - 布尔bool
- 类型限定
 - 在定义变量时有两种情况
 - 未限定，比如var kk = 1就是不限定kk是怎样类型的变量，此时这种变量可以进行赋值任意类型的数据，比如 kk = True作为后续修改值是可以的
 - 限定，比如

```
var kj :int= 1
```

 在后边加上冒号和数据类型比如int，那么就限定了kj这个变量的类型只能是int，后续对kj作赋值时只能赋值int型和float型而不能是其他的类型
 - 比如kj = 2.6的话，那么kj会直接取整为2
 - 但是如果kj为float且初始为1.4，那么kj = 4那么就是4
- 关于数据的计算
 - 字符串仅作+运算，意思是拼接字符串
 - 数字的类型比如int和float就是正常四则运算
- 数据类型的转换
 - 数字转字符串：比如String.num(1.1)转换小数，和String.num_int64(1)转换整数
 - 数字与布尔值：限定为布尔类型的变量如果赋值了非0数字，那么就相当于赋值为true，反之为false
- 逻辑运算符：! -取反，
- 其他值类型变量

- 例子, var a = 1, var e =2
- **值变量**: 执行a = b时, 仅仅是将b的值传给a而不对b值产生影响
- **引用变量**: 执行 a = b 时, 会将b这个变量传给a, a的值的修改会影响到b
- 多维变量
 - 比如二维就这样定义, var v =Vector2(114, 514), 那么这个里边的参数其实是 x, y
 - 要修改的话就是v.x = 1919, 就会把114修改为1919
- 变量除定义的过程外, 必须在函数内进行数值的修改, 内置代码中的许多变量在修改后, 会直接对游戏内容产生实质性的影响。而自定义的变量则不会。因此, 可以用自定义的变量来定义逻辑上的游戏数据, 而后通过判断逻辑上的数据情况, 来调整内置代码中的变量, 对游戏的画面、音效等产生实际的影响。
- 关于一些内置函数:

• `void _ready() virtual`

当节点“就绪”时被调用, 即当节点及其子节点都已经进入场景树时。如果该节点有子节点, 将首先触发子节点的 `_ready()` 回调, 稍后父节点将收到就绪通知。

对应 `Object._notification()` 中的 `NOTIFICATION_READY` 通知。另请参阅用于变量的 `@onready` 注解。

通常用于初始化。对于更早的初始化, 可以使用 `Object._init()`。另见 `_enter_tree()`。

注意: 该方法对于每个节点可能仅调用一次。从场景树中移除一个节点后, 并再次添加该节点时, 将不会第二次调用 `_ready()`。这时可以通过使用 `request_ready()`, 它可以在再次添加节点之前的任何地方被调用。

• `void _process(delta: float) virtual`

在主循环的处理步骤中被调用。处理发生在每一帧, 并且尽可能快, 所以从上一帧开始的 `delta` 时间不是恒定的。 `delta` 的单位是秒。

只有在启用处理的情况下才会被调用, 如果这个方法被重写, 会自动进行处理, 可以用 `set_process()` 来开关。

对应于 `Object._notification()` 中的 `NOTIFICATION_PROCESS` 通知。

注意: 这个方法只有在节点存在于场景树中时才会被调用 (也就是说, 如果它不是孤立节点)。

- 关于函数
 - 也叫方法和模块, 可自定义也有内置
 - 一个函数包含名称, 输入信息, 处理流程, 处理结果等
 - 函数可以指定调用

```

func _enter_tree():
    >I    homo()
    >I

func homo():
    >I    print('我浪, 这么厉害')
  
```

- 比如 这里就是

对homo自定义函数的内部调用，不加self也可以

- 注意return是一个函数的非必要的结束标志，它可以返回一个数据
- 自定义函数时和python一样可以加上需要传入的参数（形参），同时呢，这个形参可以指定其类型

```
▼ func _enter_tree():  
  >I homo('我浪，这么厉害', 1)  
  >I  
▼ func homo(s:String, j:int):  
  >I print(s+String.num(j))
```

- 比如
- 还可以加上默认值，和python一样的
- 哦对了，经常看到有这么个写法

```
▼ func _exit_tree() -> void:  
  >I pass
```

- 这个是函数返回值的类型转换，这里是把返回值转换成无，就是不返回
- 内置虚函数：没有实际处理流程的函数，可由制作者自行编写，节点内的内置虚函数会在特定条件下触发运行

条件 1：当节点本身或其周围节点状态发生改变时。【节点创建、节点入“树”、节点出“树”、节点死亡前、节点的子节点全部加入场景树】

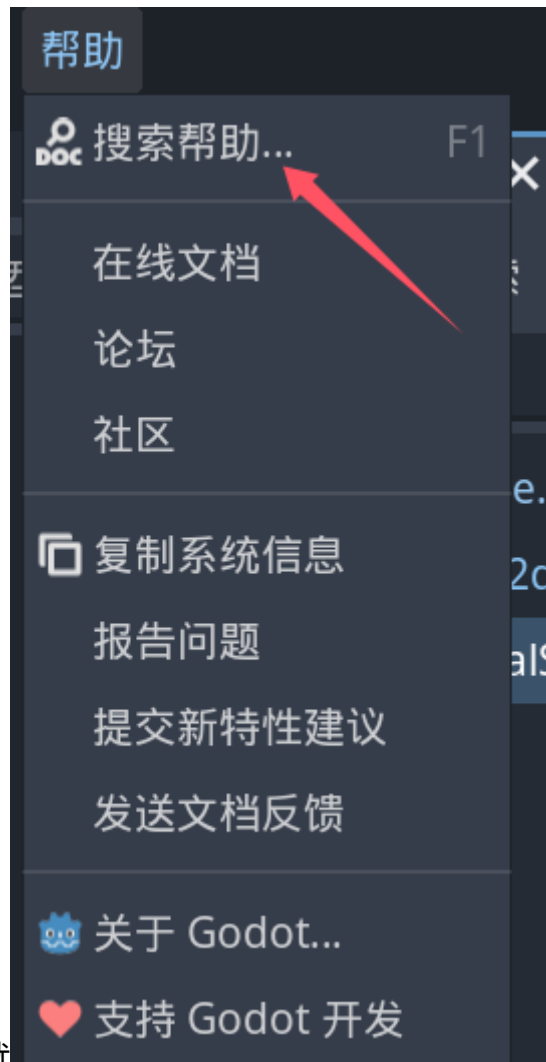
条件 2：当场景树的状态发生改变时，且节点自身处于“树”下。【画面刷新、物理引擎刷新、硬件设备输入】

条件 3：其他情况。

- 常用虚函数：_init, ready, _process, s

- 具体函数的介绍可以对这个函数名字的地方ctrl+单击查看
- 然后捏，变量有局部与全局之分，和python，c语言差不多
 - 但是，连if里边定义的变量拿出来了还是不能在if语段外边用，其他情况同理
- 一旦函数运行结束，变量本身会销毁，但相当于是装东西的容器没了，但是里头的东西还在
- if elif else：和python一样写法，但是要在函数内部用。
 - 但似乎有个神必0.3和python一样会出问题，就是说0.1+0.2在程序上不等于0.3
 - 那么可以用is_equal_approx函数来判断是否相等，把需要比较的两个参数放参数表里就行
- for循环：一般和数组一起用，和python一样的写法
- while：和python一样
- 注释写法应该也和python差不多
-
- ==注意：

- true和false不需要首字母大写
- ifelse和while都会产生自己的变量作用域，凡是if和while内定义的变量，拿出来都不能用，否则报错
- 有个东西不知道怎么用怎么办



- 那就用这个看就行了
- 关于逻辑运算true和false的判定
 - 字符串为空-false
 - 0-false
- 数组：和c里边的概念类似，但是实际和python的列表差不多

- ```
var a :Array = []
```

 这样定义就行，其实和限定类型差不多，把数组Array也当成一种类型就行，但是捏

- ```
var a :Array[int] = []
```

 如果这么写的话就是限定这个数组内的元素只能是数字，但这里呢，写个小数进去它会默认给你取整，然后转换成整型存储在对应位置上

- ```
var a :Array[int] = [1.1]
print(a[0])
```

 这个就是返回1

- 元素的访问和python从差不多，甚至-1也表示倒数第1个元素

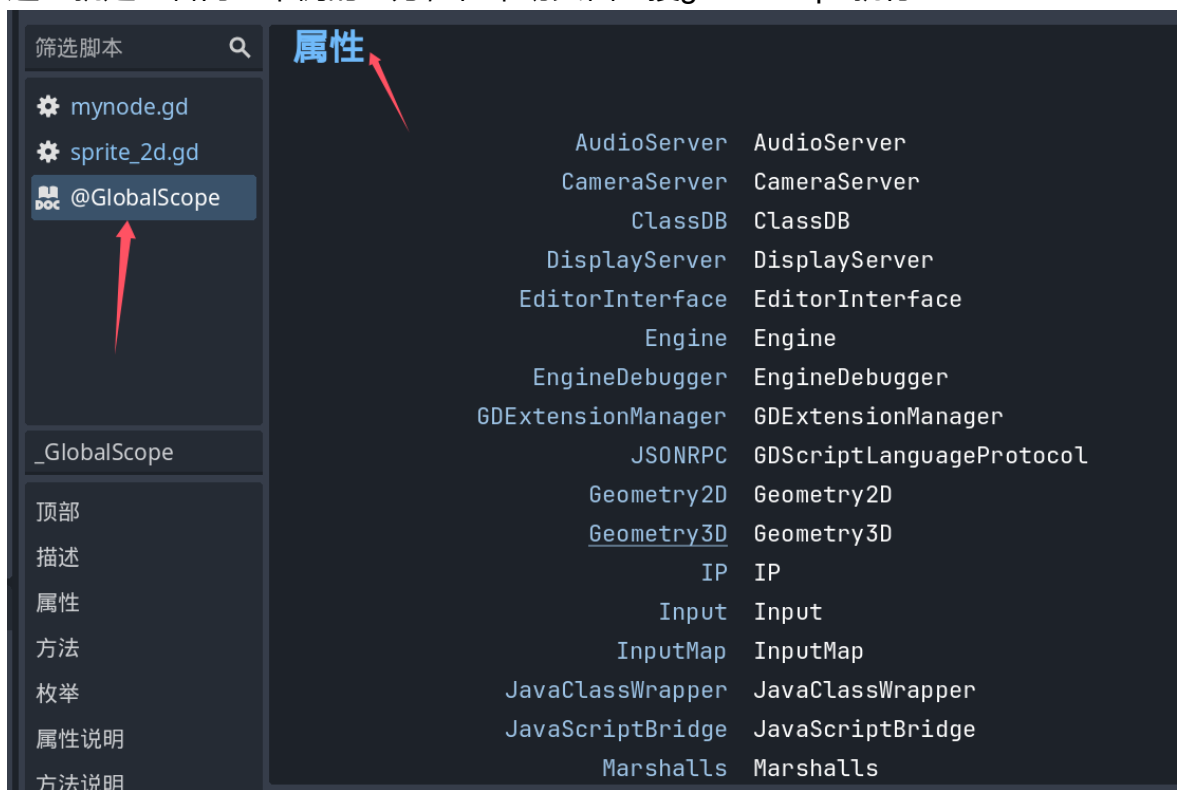
- 一些方法
  - append, 从尾部添加指定元素
  - erase, 删除检索到的第一个指定的元素
  - size, 和python的len一样
- 注意:
  - 和python不同的是, 数组可以直接通过赋值 像 `a = b` 这样把b数组全部赋值给a变量, 此时这个a变量是引用变量, 引用了b变量存储的数组数据,
  - 当a和b都在统一作用域时, 对a的改动会直接影响到b的数组数据
  - 但是如果b是外部/全局变量, 那么a的改动不会改变b

## • 单例:

### ①单例

- 单例是一个可以在任何一个脚本中对其进行直接访问的对象, 分为内置单例与自定义单例。每个单例都是独一无二的对象。
- 内置单例不是节点, 主要成员是各类 Server, 开发者可以使用它们直接控制游戏程序的图形与音效等内容。此外, 还包括了一些其他对象, 它们涉及的范围包括网络、时间、电脑系统、输入等。
- 自定义单例必须是节点类型的对象, 是开发者自定义的全局对象。

- 这里就是查看内置单例的地方, 在帮助文档里搜globalscope就行

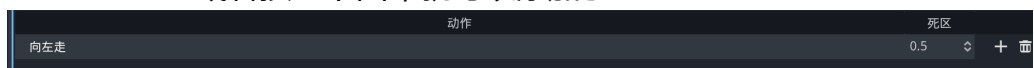


- 首先是input单例: 可以对玩家的按键情况做出反馈
  - action (可交互的键位) 的手动设置:

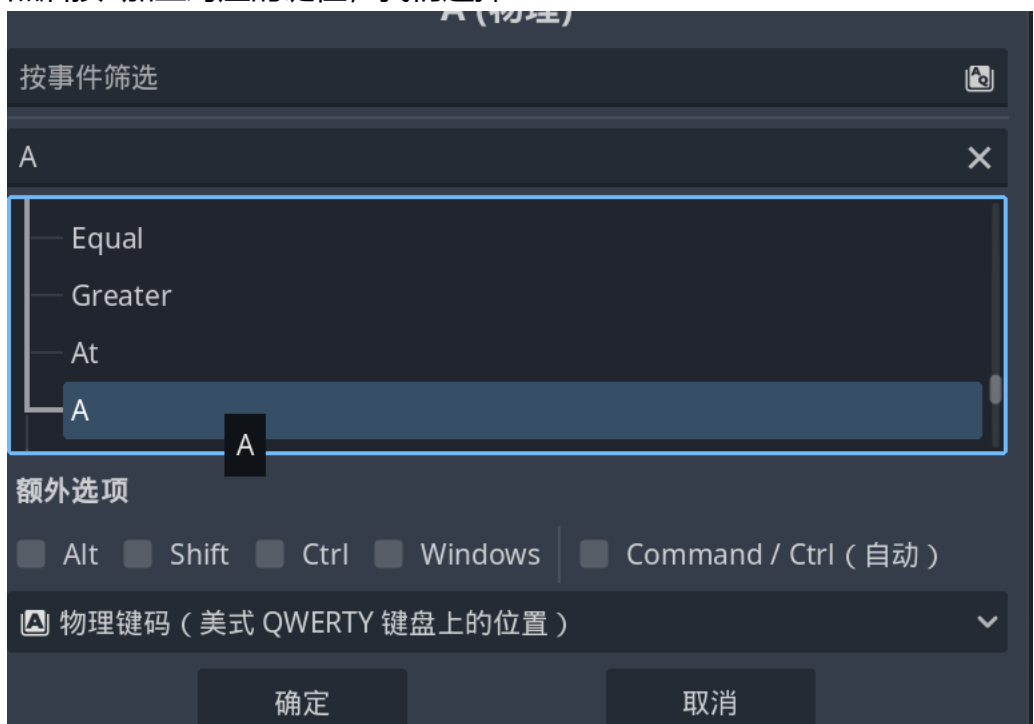


这里就可以设置了

- 例子：比如添加“向左走”在这里输入名称 **添加新动作** 再点击 **+ 添加** 或者按一下回车就可以添加了



- 然后按+加上对应的键位，我们选择A





然后确定就行

- 脚本编写
  - 然后我们在需要控制的节点编写一下脚本

```
func _ready() -> void:
 >I print(Input.get_action_strength('向左走'))
 >I
 >I pass
```

• # Called every frame, 'delta' is the elapsed time since the previous frame.

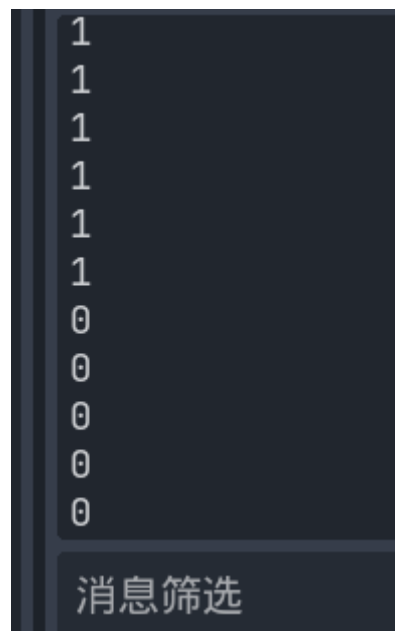
这里选择Input示例然后用get\_action\_strength方法，参数里写上我们刚才设定的action名称（字符串），然后运行，那么就会返回一个0，因为没按A嘛

- 但是我们在\_process函数里加上这串代码再按A就会返回1了

```
func _process(delta: float):
 >I print(Input.get_action_strength('向左走'))
 >I pass
 >I # self.position.x = self.position.x+1
```

这个函数是随画面的刷新而触发的，基本上一直都在触发，就像一个死循环

- 然后按A就会实时在下方的终端里变化1和0了



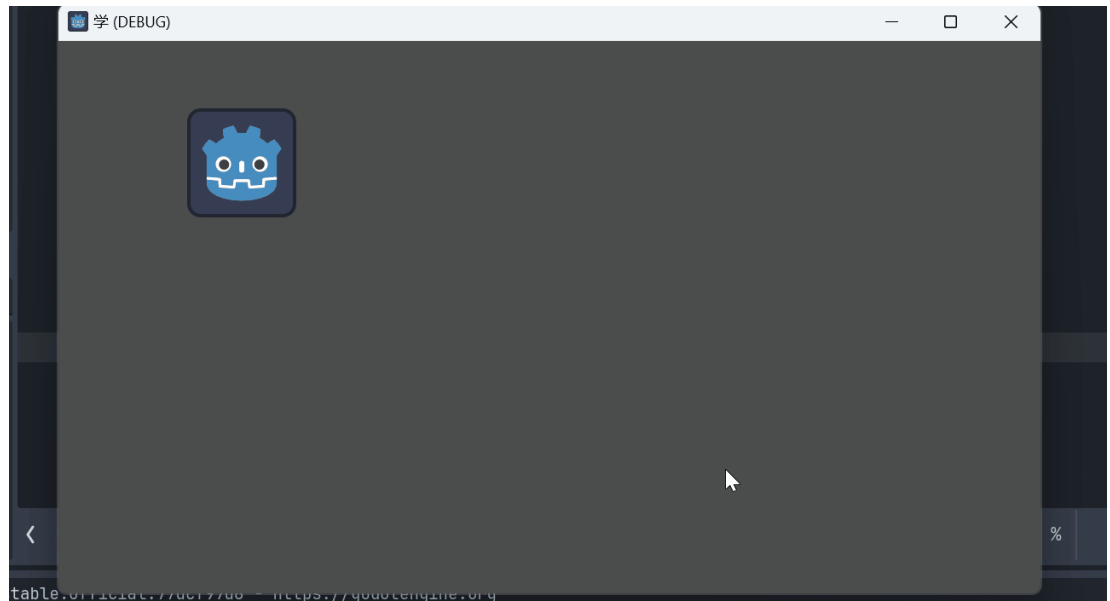
```
1
1
1
1
1
1
0
0
0
0
0
消息筛选
```

- 这里就涉及一种机制——轮询，一秒内数十次检测游戏输入情况的编码方式
- If+轮询+修改内置变量=游戏在玩家的控制下发生实质性的改变

- 比如

```
func _process(delta: float):
>I if Input.get_action_strength('向左走'):
>I >I self.position.x = position.x-1
>I >I self.position.y = self.position.y-1
>I |
>I pass
```

效果



- 自定义单例:

### 三、自定义单例

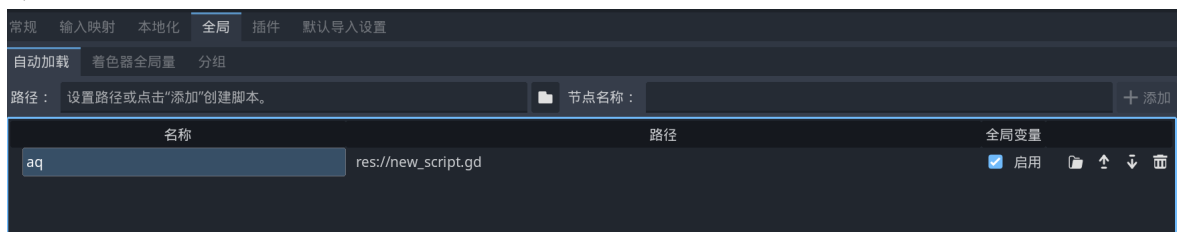
#### ①自定义单例步骤

- 在项目设置中选择 Autoload。
- 选择脚本路径
- 点击添加

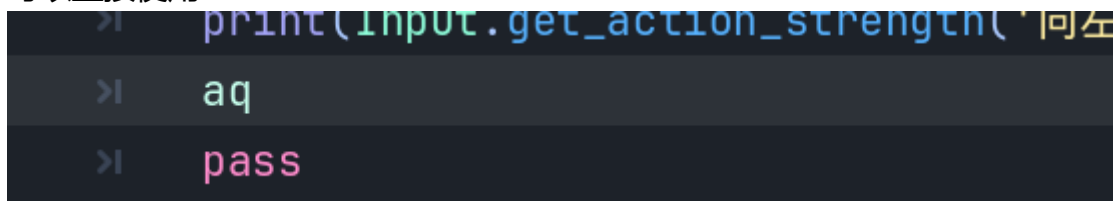
#### ②自定义单例特征及用途

- 可以在任意一个脚本中对它们进行直接访问
- 用于记录全局变量

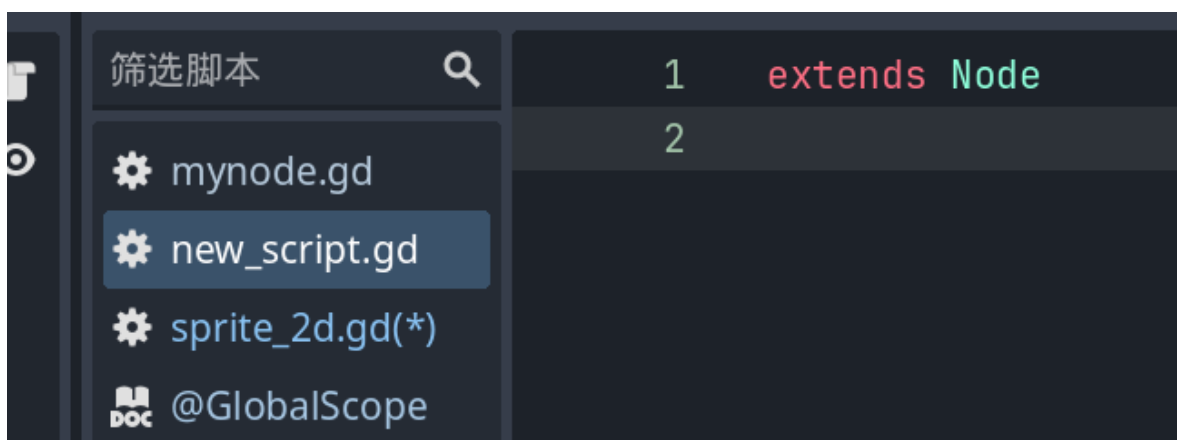
- 在选择全局前先在文件区域新建一个脚本然后在项目设置里找全局设置就行
- 效果



- 可以直接使用



- ctrl+单击点进去可以编写脚本



```

 extends Node
 func akkl():
 print(114514)

```

- 比如

然后

```
aq.akk1()
```

- 就可以看到

```

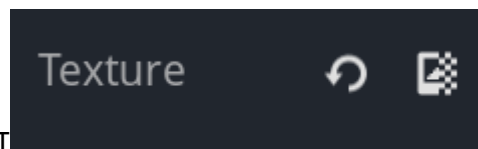
0
114514
--- Debugging process stopped ---

```

- 

## 常用节点介绍与使用

- Sprite2D：用于显示图片



- 创建后，将图片拖拽至右侧的texture属性即可

```

1 extends Sprite2D
2
3
4 # Called when the node enters the scene tree
5 func _ready() -> void:
6 pass # Replace with function body.
7
8
9 # Called every frame. 'delta' is the elapsed
10 func _process(delta: float) -> void:
11 pass
12

```

- 

对该节点创建脚本后可以看到有这么几个函数

- 其中\_process函数会在每次游戏画面刷新后执行一次
  - 我们对\_process函数进行修改

```
self.position
```

- 写上这个后，这个position其实是一个内置变量，可以通过按住ctrl后再单击这个变量可以看到其的参数介绍

- **Vector2 position** [默认: Vector2(0, 0)]  
set\_position(值) setter  
get\_position() getter

位置，相对于父节点。

- **float rotation** [默认: 0.0]  
set\_rotation(值) setter  
get\_rotation() getter

旋转，单位为弧度，相对于该节点的父节点。

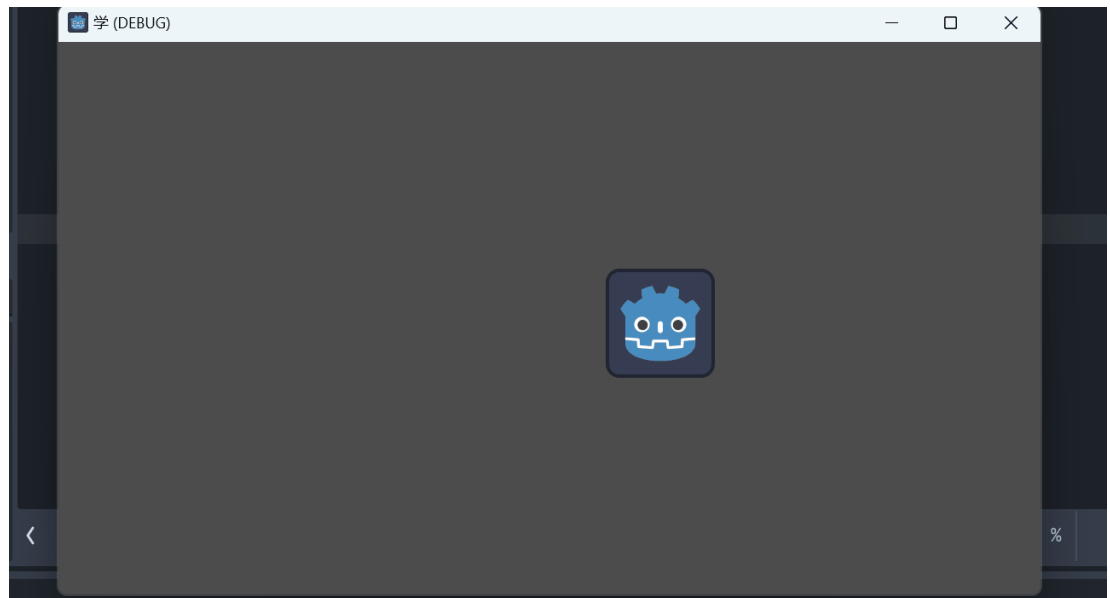
注意：这个属性在检查器中是以度数编辑的。如果你想在脚本中使用度数，请使用 rotation\_degrees。

- 这里注意到position是一个二维向量，那么就可以修改其x, y值

- ```
self.position.x = self.position.x+1
```

改成这个后

运行看看



- **Label**: 用于显示纯文本的节点

```

extends Label

# Called when the node enters the scene tree
▼ func _ready() -> void:
    >| pass # Replace with function body.

# Called every frame. 'delta' is the elapsed
▼ func _process(delta: float) -> void:
    >| pass

```

- 创建脚本后长这样

- 类：**  **Label**

继承：  Control <  CanvasItem <  Node <  Object

派生： → "label.gd"

用于显示纯文本的控件。

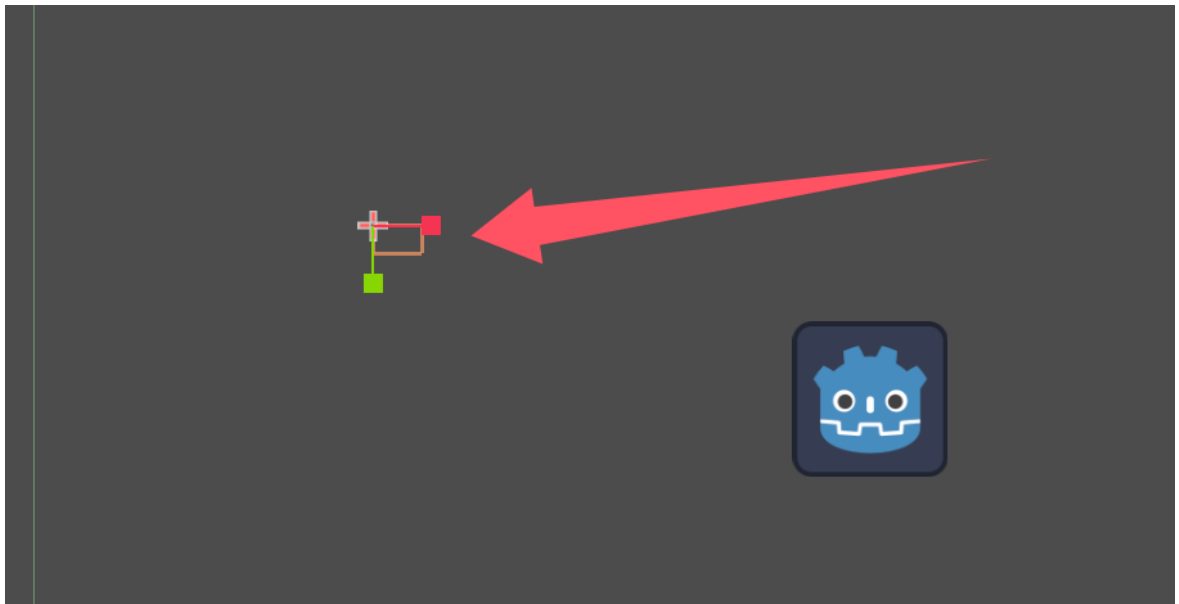
描述

用于显示纯文本的控件。可以控制水平和垂直对齐方式以及文本在节点包围框内的换行方式。不支持粗体、斜体等富文本格式。这种需求请改用 RichTextLabel。

在线教程

 - [2D Dodge The Creeps 演示](#)

属性



这个就是文本的位置，可以调整的

```
var c:String= 'Hello World!'
```

```
▼ func _ready() -> void:  
    > self.text = c
```

写成这样的就是给这个文本节点设置

文本的内容，效果是

