**让我们开始吧**

在开始之前，请确保您已经有了一定的C#基础

C#是一种面向对象、强类型和静态类型的编程语言。这与Lua有很大的不同，Lua是一种动态的、弱类型的编程语言，与C#相比，限制较少，编码习惯也有些许不同。

因此，强烈建议您从一个清晰的角度开始学习C#。下面是一些可以开始学习的优秀资源。

**微软官方文档**

对于初学者来说，微软的官方文档和教程可能是一个不错的选择。（以下为超链接，按住ctrl并点击可访问）

* **文档[目录](https://learn.microsoft.com/zh-cn/dotnet/csharp/)**
* **[交互式Hello World](https://learn.microsoft.com/zh-cn/dotnet/csharp/tour-of-csharp/tutorials/hello-world)**
* **[c#之旅](https://learn.microsoft.com/zh-cn/dotnet/csharp/tour-of-csharp/)**
* **[微软的“学习c#”](https://dotnet.microsoft.com/zh-cn/learn/csharp)**

**SoloLearn**

SoloLearn有一些免费的C#教程。但你可能会在网站上看到广告。（以下为超链接，按住ctrl并点击可访问）

* **[初学者C#教程](https://www.sololearn.com/learn/courses/c-sharp-introduction)**

**LearnCS.org**

LearnCS.org是一个很好的学习网站。 它是完全免费的，但是它更像文档风格，而且有广告。总的来说，当你可能需要快速参考的时候，它仍然是一个伟大的网站。

当然，你也可以通过其他的途径来学习C#，你可以通过自己的方式，自己喜欢的模式来一步步学习，我们是不强求的。

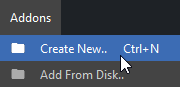
关于下文各个api，你可以去超链接内看，我也会在文章的最后进行大部分的翻译

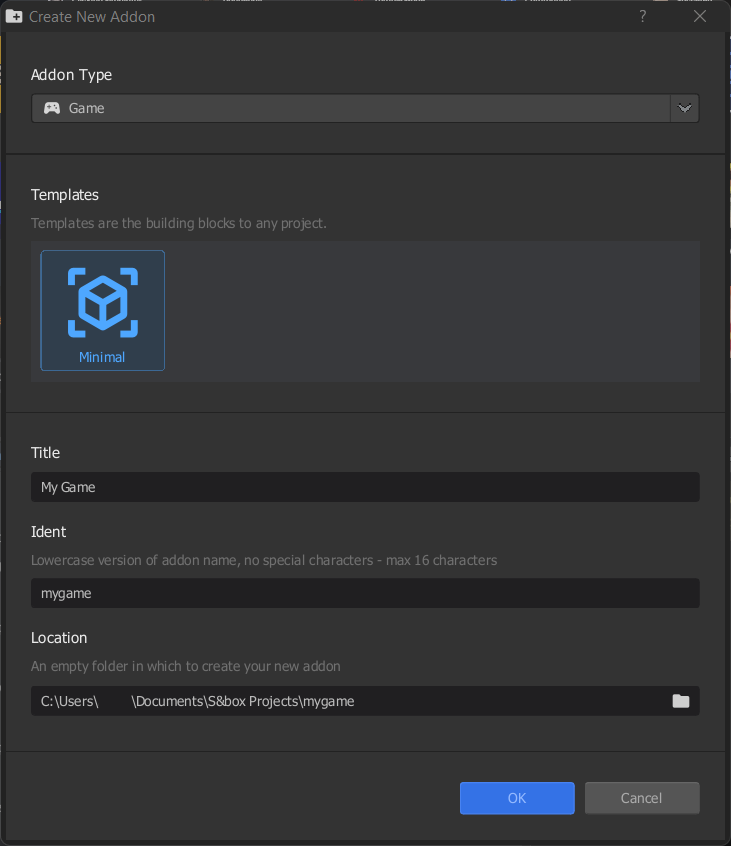
**创建游戏**

在创建游戏之前，你应该有一个代码编辑器，最好是支持C#的编辑器，最简单也是目前比较推荐的是设置Visual Studio，但你也可以设置其他的编辑器。（甚至于记事本）

**创建一个小游戏**

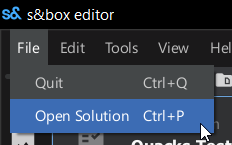
首先打开你的sbox开发版，依次点击Project-> Create New... 创建一个新的游戏项目，你可以把它放在你磁盘上任何地方。





**代码**

创建好游戏后，你可以通过你的Visual Studio解决方案访问你的游戏模式代码，最简单的方法是通过你的编辑器打开它。



默认情况下，你创建的模式只有最基本的游戏代码，可以通过鼠标左键射出一个Terry。

**内容**

你的所有内容都可以放在addon文件夹内的文件夹里。

* addons/myaddon/config -用于定义自定义资产类型
* addons/myaddon/fonts - 这里的字体会自动加载并可在用户界面中使用。
* addons/myaddon/materials- 贴图文件
* addons/myaddon/models- 模型文件
* addons/myaddon/particles- 粒子特效
* addons/myaddon/shaders- 着色器文件
* addons/myaddon/sounds- 音效文件

**运行游戏的多个实例**

你可以运行多个游戏实例来测试多人游戏功能，而不需要其他用户。

要做到这一点，在启动游戏之前，s&box的两个实例都需要执行sv\_lan 1

接下来，托管客户端应该启动一个本地游戏。

然后，只需在非主机客户端的控制台中输入connect localhost（在非工具模式下按F1键查看控制台）即可连接。

**需要注意的事项**

你的客户端在所有情况下都是一样的（名字、玩家ID等）。

你将不得不从自己那里下载你的游戏，这可能需要一些时间。

**设置Visual Studio编辑器**

**下载并安装visual studio 2022**

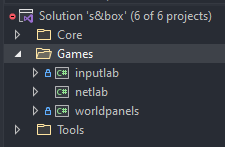
在微软官网中，下载[visual studio 2022社区版](https://visualstudio.microsoft.com/zh-hans/downloads/)，并安装.NET

如果你需要额外的指导，请参考[官方指南](https://learn.microsoft.com/zh-cn/visualstudio/install/install-visual-studio?view=vs-2022)。

**打开你的项目**

最简单的方法是从编辑器打开或( CTRL + P )

你的解决方案包含每个代码插件(游戏，工具，插件)-每次你创建一个新的插件，它会自动重新生成。



**故障排除**

如果由于依赖项无法加载而导致大量错误，请确保您已安装了.NET Framework 7和最新版本的Visual Studio 2022。

**设置Rider编辑器**

如果您认为visual studio不和您胃口，您也可以使用Rider

**获得Rider**

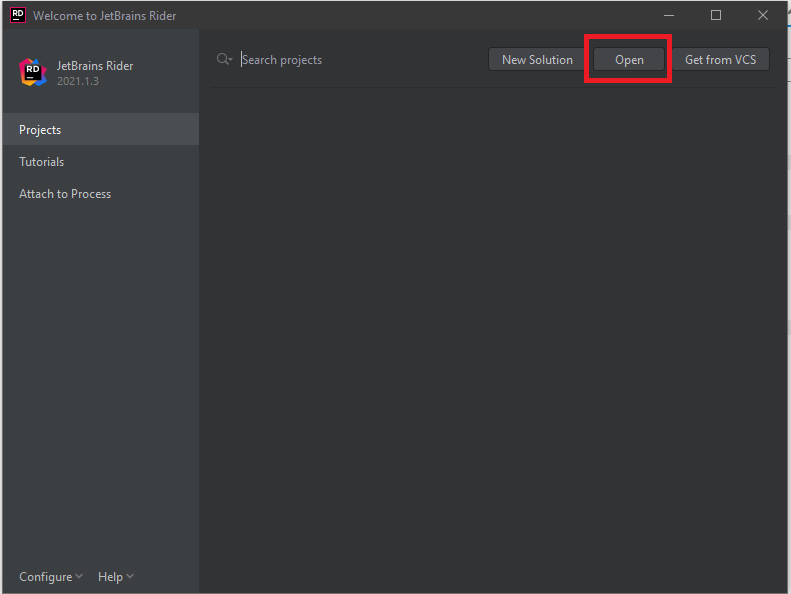
在该[链接](https://www.jetbrains.com/rider/)下载或购买Rider

**解决方案**

要获得.sln文件，您需要启动游戏。你的s&box.SLN应该在根目录中。

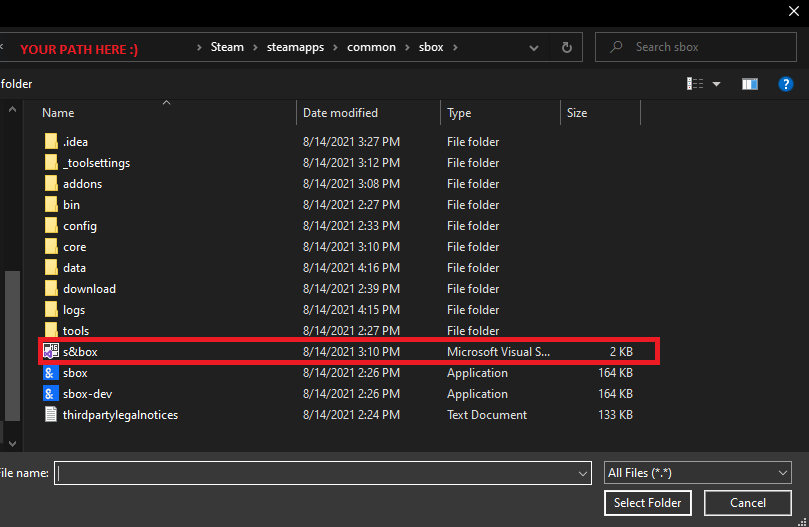
[请注意，在你每次向你的游戏添加插件时，你都必须这样做，然后才能进行编辑。]

1. 点击打开

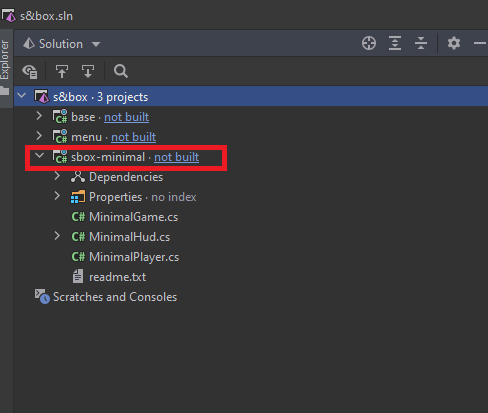


2.到你的游戏文件夹下

3.选择S&box.sln



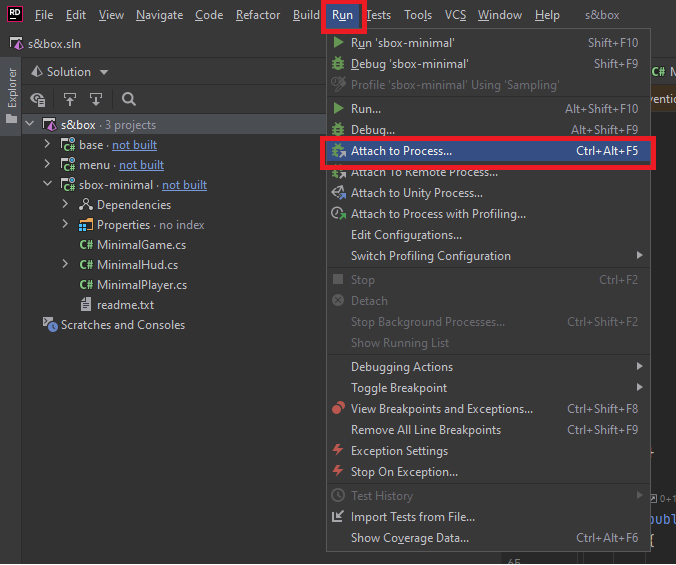
4.现在你应该可以看见你的插件了



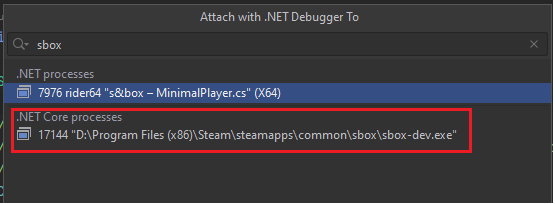
**调试**

1.启动你的S&box-dev，并且进入你的游戏模式

2.在rider中依次点击Run -> Attach to process...



3.选择S&Box程序



**故障排除**

如果你在所有方面都出错，请确保你安装了.NET 7 SDK。

如果你从本应生成代码的东西上得到随机错误，请确保你启用了Roslyn分析器。

**游戏环路**

**每一Tick**

**模拟图**

****

参考：

<https://asset.party/api/Sandbox.Game.BaseGameManager.Simulate(Client)>

<https://asset.party/api/Sandbox.Entity.Simulate(Client)>

每个Tick（默认情况下每秒60次Tick），客户端将向服务器发送一条命令。这个命令会被模拟。这意味着它从命令中获取输入（即，新的ViewAngles）并将其应用于玩家(pawn)（即，将旋转设置为ViewAngles）

当这个命令被发送到服务端的时候，客户端也在本地运行着同样的命令。这就是所谓的预测。这使得像行走、射击和跳跃这样的行为，看起来没有网络延迟

在客户端运行命令后，它将存储所有改变的实体变量，当服务器返回这些实体变量时，它会比较它们。如果它们不同，那么就会发生预测错误。

**每一帧(Frame)**

这些东西在客户端上每一帧都被调用。

**构建输入**

****

参考：

<https://asset.party/api/Sandbox.Game.BaseGameManager.BuildInput()>

<https://asset.party/api/Sandbox.Entity.BuildInput()>

每一帧我们都建立了输入端。这将会把来自鼠标、键盘和控制器的输入转换为一个命令，可以在下一Tick发送到服务器。

在BuildInput中，你可以使用Input函数来处理输入--例如，使用Input.ClearButtons()来清除按钮--以及为标有[ClientInput]的属性赋值。

这是通过将InputBuilder类传递给游戏模式来实现的--游戏模式再将其传递给当前的摄像机和玩家(Pawn)。这通常用来做一些事，比如将鼠标移动改变为视图角度。

你可以通过覆盖BuildInput在你的游戏类中覆盖所有这些行为。

**帧模拟**



参考：

<https://asset.party/api/Sandbox.Entity.FrameSimulate(Client)>

<https://asset.party/api/Sandbox.Game.BaseGameManager.FrameSimulate(Client)>

因为模拟只发生在每一帧（默认为60fps），而玩家的显示器可能会显示比这高得多的帧率，所以像视图方向这样的东西在客户端会感觉很不稳定。为了解决这种不匹配，FrameSimulate在每一个客户端帧都会被调用，以便给游戏一个机会来平滑任何可能出现的停滞。

这通常被用来更新基于视图方向的东西（比如Pawn.AimRay）。

重要的是，你必须意识到这只是客户端，如果你在这里设置了什么，那么它必须在模拟中更新。

**文件系统**

标准的.NET文件访问受到限制，这是为了防止恶意访问您的文件。这意味着您不能直接使用System.IO.File。

相反，Sandbox为几个虚拟文件系统提供了一个[BaseFileSystem](https://asset.party/api/Sandbox.BaseFileSystem)，这些虚拟文件系统只能访问特定游戏目录中的文件。

// 使用BaseFileSystem读/写文件的示例

if ( !FileSystem.Data.FileExists( "player.txt" ) )

FileSystem.Data.WriteAllText( "player.txt", "hello world" );

var hello = FileSystem.Data.ReadAllText( "player.txt" );

// 使用File System保存和加载玩家数据的例子。

class PlayerData {

// ...

public static void Save(PlayerData data){

FileSystem.Data.WriteJson("player\_data.json", data);

}

public static PlayerData Load(){

return FileSystem.Data.ReadJson<PlayerData>("player\_data.json");

}

// ...

}

关键笔记：将一个类传递给FileSystem.Data.WriteJson，变量必须是对象属性，例如:

public class PlayerData {

public int playerLevel { get; set; }

public int playerMaxHealth { get; set; }

public string playerUsername { get; set; }

}

### **FileSystem.Mounted**

FileSystem.Mounted是一个集合了核心游戏、当前游戏模式及其依赖性的所有加载内容的文件系统。它是以下目录的组合:

* sbox\core\
* sbox\addons\myaddon\
* sbox\addons\myaddon\code\
* sbox\addons\base\
* sbox\addons\base\code\
* sbox\addons\citizen\
* sbox\addons\citizen\code\
* sbox\addons\rust\
* sbox\addons\rust\code\

任何时候你试图读取一个文件，它都会搜索这些路径。

### **FileSystem.Data**

[FileSystem.Date](https://asset.party/api/Sandbox.FileSystem.Data)是存储游戏用户数据的地方。

C:\Steam\steamapps\common\sbox\data\org\game\

### **FileSystem.OrganizationData**

[FileSystem. OrganizationData](https://asset.party/api/Sandbox.FileSystem.OrganizationData)用于存储组织中多个游戏的用户数据。

C:\Steam\steamapps\common\sbox\data\org\

**事件系统**

事件系统类似于Garry's Mod中的Hook系统。它允许你在类上注册函数以接收特定全局事件的回调。

**设置你的类**

实体和面板会自动为你注册/取消注册，对于其他的东西，你只需要调用Event.Register和Event. Unregister

public class TeamManager {

public TeamManager() {

Event.Register( this ); //在不是实体或面板的东西上执行此操作

}

[Event.Tick]

public void OnTick() {

Log.Trace( "blah" );

}

~TeamManager() {

Event.Unregister( this );

}

}

**注册一个回调函数**

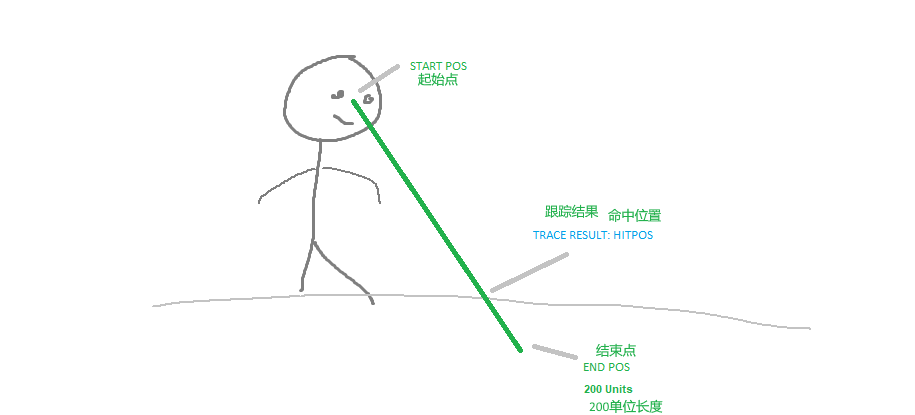
**路径跟踪(Traces)**

**何为路径跟踪？**

路径跟踪实际上是一条虚构的线。当你运行了一次Trace，他就会返回一个Trace Result

来告诉你这条线击中了什么东西。

例如，玩家想要生成一个箱子。你可以从玩家的眼到玩家所看方向的200个单位运行一条Trace。Trace Result会显示它击中了一个点，而箱子会生成在那个点上。



**跟踪**

路径跟踪是使用Trace静态函数构建的，并使用成员进行配置。你可以使用Run( )来运行跟踪并获得结果(Trace Result)。

var mytrace = Trace.Ray( startPos, endPos ); //创建trace，并设置起始与结束点

mytrace = mytrace.WorldOnly(); //仅世界

var result = mytrace.Run(); //运行trace

你也可以直接在创建完的同时运行

var tr = Trace.Ray( startPos, endPos ).Run();

或者这样

var tr = Trace.Ray( startPos, endPos )

.Ignore( playerEntity )

.Ignore( playerVehicle )

.Size( 10 )

.Run();

**大小**

在上面的例子中，Size描述了要跟踪的AABB的大小

这意味着您不再是描一条简单的线，而是沿着这条线描一个更大的立方体

**运行所有**

您可以使用RunAll()函数，而不是用Run()函数结束跟踪。

使用RunAll将会让你的Trace不在第一个被击中的实体处停止。它将返回一个数组，其中包含沿途所有被命中实体的TraceResult。

该函数的一个示例用例是穿透物体的子弹。

//将返回TraceResult[]，您可以进行迭代

var tr = Trace.Ray( startPos, endPos ) .RunAll();

**Trace Result**

跟踪结果(Trace Result)是一个比较简单的结构体。它会给你关于击中的信息。

在这里你可以看到Trace是否命中并使用命中位置生成一个高于它10个单位的西瓜。

if ( tr.Hit ) {

var ent = new ModelEntity();

ent.SetModel( "models/sbox\_props/watermelon/watermelon.vmdl" );

ent.Pos = tr.EndPos + tr.Normal \* 10;

}

**机器人(Bots)**

**什么是机器人？**

机器人是可以添加到游戏内的假客户端，将占据正常的玩家槽位。它们的输入和行为可以像c#中大多数其他东西一样进行编写。

**默认机器人**

默认情况下，控制台命令 bot\_add 将添加一个模仿玩家行为的机器人。这个命令有一些你可以使用的附加参数:

bot\_add <clientIndex> <yawOffset> <forceCrouch>

**clientIndex** (def: 1)模仿哪个玩家。如果设置为0，它将不会模拟任何内容，保持空闲状态。

**yawOffset** (def: 180)机器人相对于目标玩家进行旋转。

**forceCrouch** (def: 0)机器人是否必须蹲下。

**自定义机器人**

要创建自定义机器人，您需要首先扩展bot类，它有两个重要的方法:Tick和BuildInput

**示例**

值得注意的是，下面使用的方法只被称为服务器端方法，不像普通的客户端方法，后者的输入在客户端构建并发送到服务器。

public class MyCustomBot : Bot {

[ConCmd.Admin( "bot\_custom", Help = "Spawn my custom bot." )]

internal static void SpawnCustomBot() {

Game.AssertServer(); //创建自定义机器人的实例。

\_ = new MyCustomBot();

}

public override void BuildInput() {

// 在这里，我们可以选择/修改机器人在每一tick的输入行为。

// 我们将通过按住PrimaryAttack按钮让它们不断进行攻击。

Input.SetButton( InputButton.PrimaryAttack, true );

}

public override void Tick() {

//在这里，我们可以在每个tick中使用机器人做一些事情。

//在这里，我们将每次打印机器人的名字。

Log.Info( Client.Name );

}

}

**Pawns**

每个机器人都有一个客户端属性，引用其底层的假客户端。每个机器人都可以有自己的pawns，否则就可以做任何其他普通客户端可以做的事情。

在我们了解了这些比较基础且有用的概念知识后，让我们来做一些更基础的事情