# Unity

## UnityWeb链接配置

https://24nyae.cn/Space\_Test/index.html?screen-orientation=0&hide\_navigate=1&hide\_status=1#SceneType@0&version@3.0.6~~&t@f73bd0d978284e6ea971ecd006a0da87&landID@Qez#p~~

http：//... 代表项目链接

screen-orientation 0横屏，1竖屏

hide\_navigate =0/1 展示或者隐藏导航栏

hide\_status =0/1 展示或者隐藏状态栏

SceneType 代表游戏初始场景的ID

version 版本号（不同于游戏真实的版本号，仅用于刷新缓存用）

token AppToken （不参与后台配置，代码层会加）

landID 土地ID （不参与后台配置，代码层会加）

#p 用于标识从网页进入

**此链接配置可以增加配置，注意格式 参数和参数之间用 " & " 符隔开**

## 个人空间当前框架

### UI框架：SUIFW

**使用详解：**UIManager管理类负责提供打开关闭面板接口：ShowUIForms(string uiFormName, bool IsRedirection = false)，CloseUIForms(string uiFormName)。

生命周期函数Awake()里会动态加载出挂载UI预设的Canvas,不同类型弹框的节点及遮罩层；调用ShowUIForms接口会动态加载出需要弹出的UI面板预设；根据传入uiFormname会读取UI配置文件UIFormsConfigInfo.json去加载对应目录下的UI预制体。



个人业务开发时创建自己的业务脚本继承BaseUIForm，BaseUIForm是所有面板类的基类，里面可自定义自己面板的UIFormType(UI窗体类型),UIFormShowMode(UI窗体显示类型)，UIFormLucenyType（UI窗体透明度类型）。

同时有Display(),Hiding(),Redisplay(),Freeze()等生命周期函数，对应打开，关闭，重新打开，冻结状态的调用。

**关于打开面板参数传递：**

目前参数传递是通过打开面板时事件配发的，打开面板时会SendMessage(string msgType,string msgName,object msgContent)把需要的数据带入，面板的Awake()或Start()生命周期函数里去做监听，监听到接收数据处理业务逻辑；

**开发规范：**

各个业务模块开发代码统一放在Scripts/SUIFW/UIScripts目录下，命名规范最好预制体与脚本XXXUIForm名字同名这样便于查找对应预设。

引用预制体放入Resources/SUIFW\_Res/UIPrefabs目录下。

## UI适配

### UGUI

1,设置canvas Scaler Mode为：Scale with Screen Size. (统一画布比例与屏幕大小)

（1）随屏幕大小改变自身大小模式，Canvas画布中其他的UI空间大小随屏幕大小改变自身大小。也就是根据真实屏幕的宽高来缩放Canvas(画布).

2，将Reference Resolution 设置为 开发时固定的分辨率，一般为（1920.1080）

3，将Screen match mode设置为 match width or height。

（1）根据真实屏幕的宽高比按指定的 Match 值来缩放 Canvas。 当match 为0时 按宽度进行等比缩放，当match为1时，按高度进行等比缩放。 目前macth值为0.5，去进行整体等比缩放。

4，获取真实屏幕分辨率的值，计算真实屏幕分辨的宽高比的系数，然后再根据开发时固定的分辨率和这个系数计算。得到一个适配的分辨率。 并把web网页的分辨率设置为此分辨率。

5，Canvas画布中的UI元素，可以使用锚点来进行适配。

## Unity web端与JS交互

### 配置

1，创建jslib文件，将jslib文件放置到Unity项目的Plugins文件夹中。在unity中c#可以通过jslib文件，实现和js的交互。

2,在jslib文件中，可以直接编写js代码接口，通过js去调用App端的接口及原生网页相关的方法。

3，C#类中定义前端响应事件（方法名相同）。

4，js中接收参数用pointer\_stringify()接受，否则传的值只是地址索引。

### JS中调用App方法

1， 用navigator.userAgent.toLowerCase()的返回值判断平台类型。

#### android平台

举例： window["AIEraJsHandler"].showNavigateStatusBar(); showNavigateStatusBar为方法名，可以直接传响应的参数。

#### IOS平台

举例：

var data = {

"methord":"changeScreenOrientation:",

"param":{orientation:"0"},

};

window["webkit"].messageHandlers.AIEraJsHandler.postMessage(data);

changeScreenOrientation 为方法名， param中传的是参数。

#### web网页

1，打开web网页： window.location.href=url;

url为网页地址。

## 触摸事件管理器

1，为了解决多个手指触摸事件冲突 （手指的 按下，按住，抬起）三个状态。 做一个触摸状态识别管理器。

核心思路为每帧记录当前的屏幕上所有touch点的touchID，再去对比上一帧所有的touchID

(1) 如果有新增，这个touch点的状态就为按下，

(2) 如果有重合，就为按住状态

（3）如果有移除，就判断为抬起状态。

详见 EashJoystick脚本

#### 摇杆

##### 单摇杆和多摇杆触摸管理

1，在触摸事件管理器内注册摇杆类，将摇杆类加入到字典中，并将touchID和摇杆类绑定。 每次touchID对应的触摸点的状态发生改变就将 触摸状态发送给摇杆类。 可支持多个摇杆。

（1）touchID 新增，将touchID和摇杆类绑定

（2）touchID 移除，将touchID和摇杆类解绑

##### 摇杆实现

1，设置摇杆的底图UI，和触摸跟随UI

2，设置触摸范围

3，设置摇杆距离中心的最大距离

4，设置摇杆的位置是否固定，如果不固定，就以手指按下的点为摇杆中心点。

5，设置触摸跟随的UI跟随手指。

6，根据跟随手指UI的偏移量，计算向量关系，获得对应可输出的向量。

具体摇杆功能类详见Joystick脚本

## 包体构建配置

### webGl

1，Unity构建包体时需要将other Serrings中的 AutoGraphicsApi关闭，手动添加 graphics Apis ：webGl 1.0.

IOS平台不支持 webGl2.0；

2，将lightmap encoding 设置为 normal Quality。 High Quality 需要webGl 2.0.目前ios 还不支持。

3,将color space 设置为gamma

3,将publishing settings 中的 decompresion fallback勾选

## 工具

### 打表工具

使用场景:

便于策划配置需求

制作流程：

策划配置对应xlsx配置，第一行描述，第二行字段名，第三行字段类型(int,float,string,bool)复杂数据类型均以string满足通过;或其他解析时切割来满足对应需求，第四行主键标识；



配置注意事项：

sheet不能重名，不能空列，最好填上默认值；

程序侧流程：

拉取最新配置，操作菜单栏GameTool->GenerateExcel生成最新配置脚本，不操作读取的还是未更新前旧的。

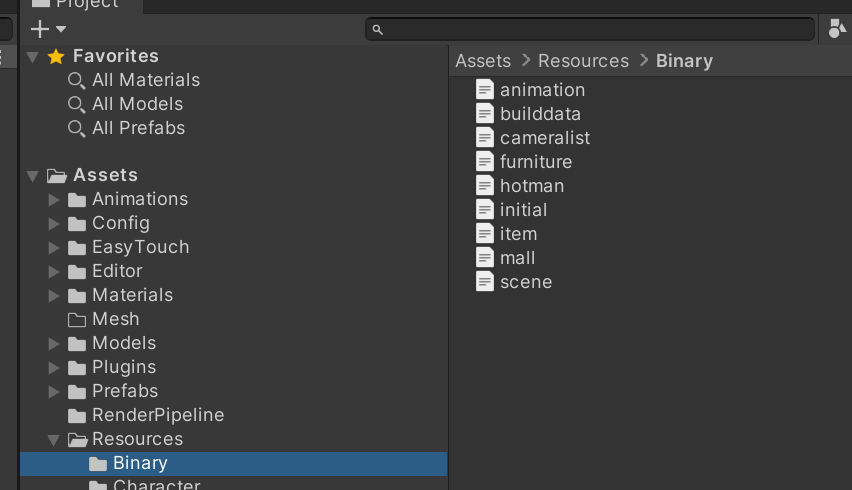
实现原理：

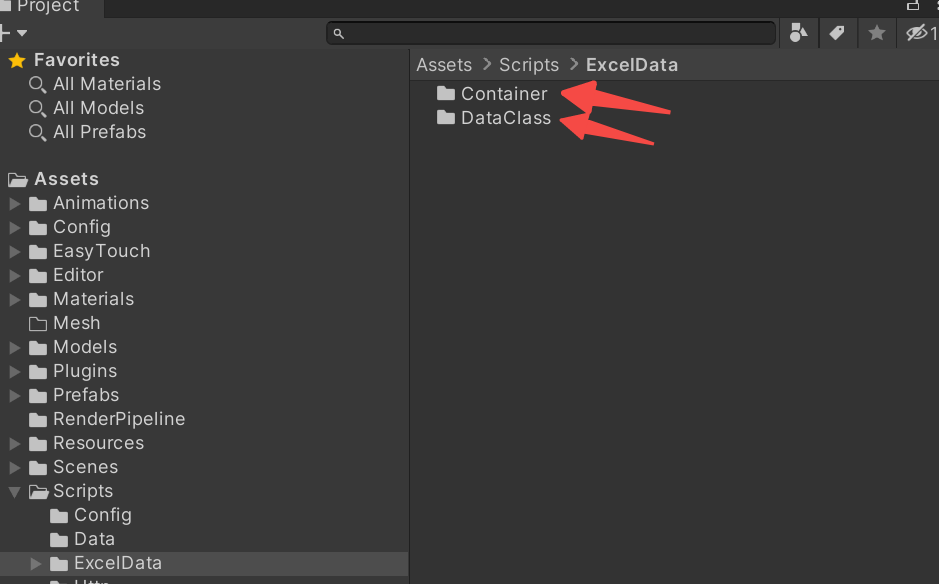
调用GenerateExcel接口会读取Assets/Config/Excel/目录下的所有.xlsx或.xls配置，通过读取文件流

生成字段对应的数据结构类，生成容器类，生成二进制数据；

数据结构类，容器类与xlsx文件同名。最终生成的这些脚本位于Scripts/ExcelData/DataClass

Scripts/ExcelData/Container目录下。

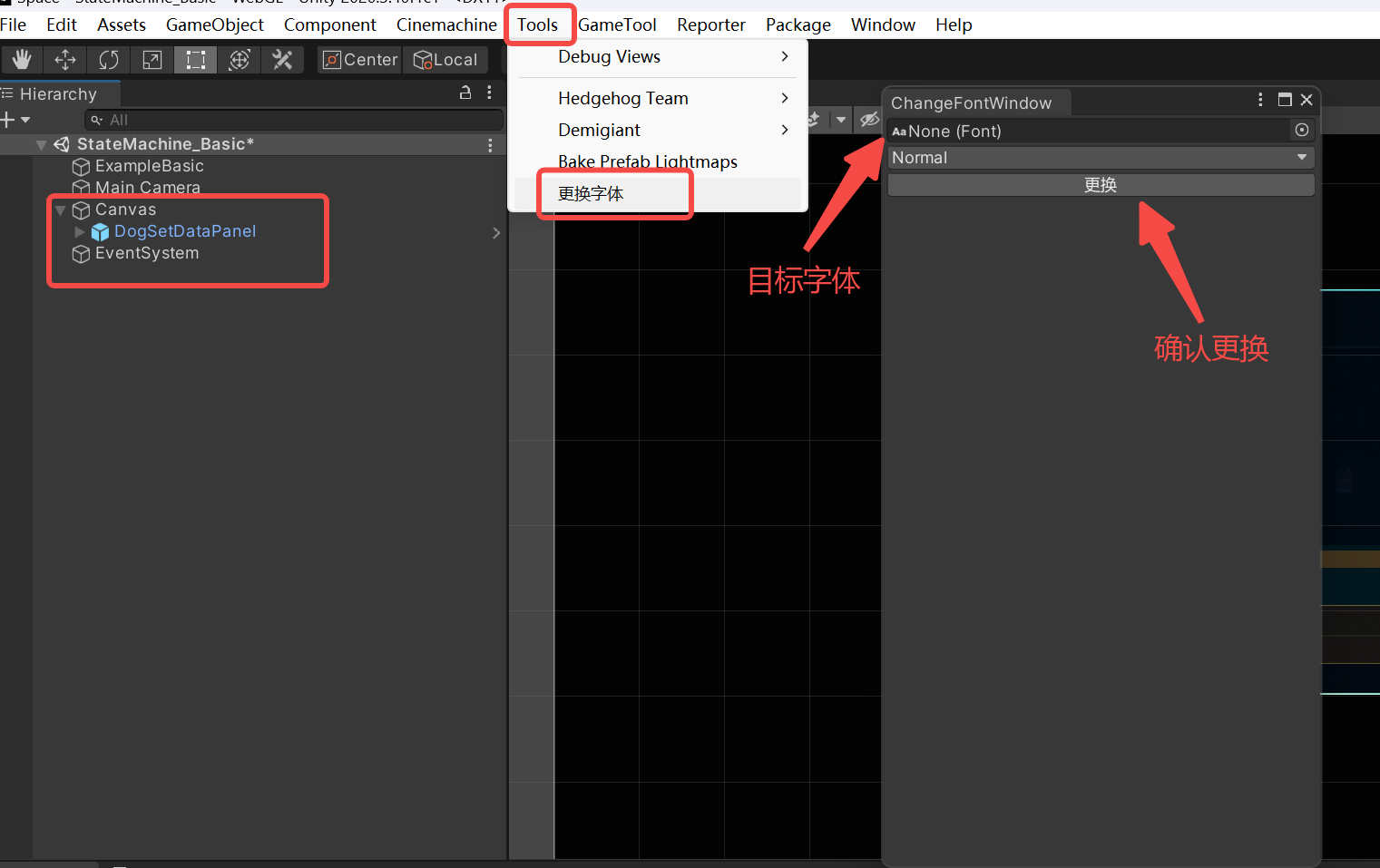




程序调用时调用BinaryDataMgr下的LoadTable<T, K>()接口就可以拿到整个数据。流程基本是读取上面生成的二进制文件，创建容器类对象，通过反射将字节流转换成对应的C#数据结构。

### 一键换字体工具

功能描述：此工具会将canvas下所有的text组件的字体，都设置为 更换的目标字体



### FigmaToUGUI

**工具入口：UnityToolBar/Tools/FigmaImporter**

1.申请Figma账号并登录

2.生成ClientCode和State

3.点击工具中OpenOauthUrl再浏览器中获取clientcode和state将ClientCode和state粘贴到Unity工具里

4.点击GetToken

5.在浏览器里Figma中找到要拼的UI 点击右键copy/Parse as/Copy Link复制链接到工具Url

6.工具中RendersPath指图集存放的目录

7.工具中RootObject为Unity/Hierarchy/Canvas节点,如果场景中没有Canvas就创建一个并设置CanvasScaler

8.工具中Font为生成文本节点使用的字体

9.点击GetNodeData，等待节点树生成

  10.在编辑完节点树之后，点击GenerateNodes按钮，在Hierarchy中就可以看到已生成好的UI节点树

**编辑节点设置ActionType的方法**：

     A.在工具中选中静态图的根节点，设置ActionType=Render, Figma会把这个节点下的子节点图片渲染到一 张图上，并在生成UGUI节点时合并成为一个Transform节点。

  B.不需要出现在生成的UGUI节点中，则设置ActionType=None。

  C.需要生成在UGUI节点中的文本，保持ActionType=Generate。

  D.工具中在ActionType列旁边的Sprite列，作用是使用已有图替换节点里的图避免出现多份。

  如：公共图集已有的图，无需再生成一份，直接将公共图集中的图拖入节点后的Sprite里就可以了

**注意1**：SVGImage不要使用，下载速度太慢会卡住工具的下载流程导致报错。

**注意2**：生成的UGUI需要自己添加组件(除了Image和Text)如：button，Scroll, toggle等需自己添加。如果要TextMeshPro代替UGUI Text组件，需要修改FigmaNodeGenerator.cs 126 useText=false,再加入TextMeshPro的安装包。

**最后**，UI Prefab 处理完后，如果需要生成代码，可以使用Tools/GenUICode。

### UIPrefab代码生成器

**工具入口：UnityToolBar/Tools/GenUICode**

**把UI的根节点拖到 GenUICode工具中选择待处理UI栏中**

**点击 查找UI控件 或者 只查找操作控件**

**点击 变量声明赋值事件注册一键完成**

**点击 生成脚本 到自定义文件目录**

**点击 挂载脚本组件 脚本会绑定到根节点上**

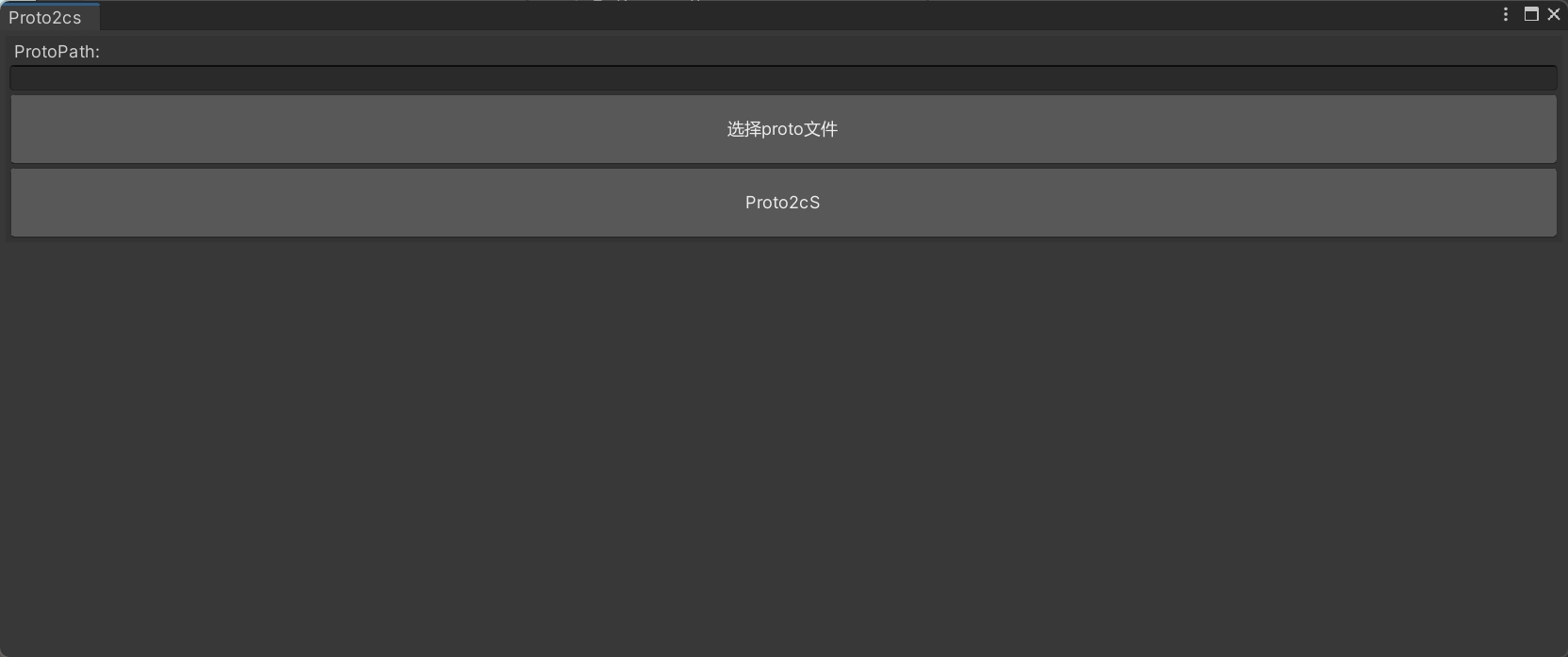
**选择UIPrefab的导出路径，并导出prefab**

**注意：自己的代码 写在//UI EVENT FUNC START以下，如变量 函数 函数实现 等等这样多次生成时会保留自己写的代码**

**其他：也可以只生成代码点击复制，粘贴到某个脚本中。**

### Proto生成工具

**工具入口：UnityToolBar/Tools/Proto2CS**



点击"选择proto文件" 找到要生成的.proto文件，在Assets同级目录下Proto文件夹下

点击"Proto2cS"生成协议脚本Assets/Scripts/ProtoMessage下

proto序列化列如: WSData wsData = new WSData(); byte[] bytes = wsData.ToByteArray();

proto反序列化如: WSData wsData = WSData.Parser.ParseFrom(byte[] bytes);

## 开发踩坑

目前网路请求都是写在协程里的，有时打开面板可能需要请求数据，请求完刷新再请求其他数据时可能这个面板关了，协程就stop了。就等不到服务器httpRequest.isSucc成功的状态，这一点处理业务逻辑时要注意

Unity发布成WebGL后生成的目录下的**StreamingAssets**文件使用**Application.streamingAssetsPath**是无法读取到的，只能通过**UnityWebRequest** 或将文件发在**Resources**目录下**，这个前期要考虑好。因为刚开始打表生成的二进制文件就放在了这里发布完测试读不到！**

浏览器对于https链接会自动屏蔽不安全的http链接，就是"**Mixed Content**"，https页面里面动态引入http资源是，会被直接block掉。 解决方法在页面的 head里加入 "**<meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests">**"代码，此配置自动将http的不安全请求升级为https。

## Unity 配置相关

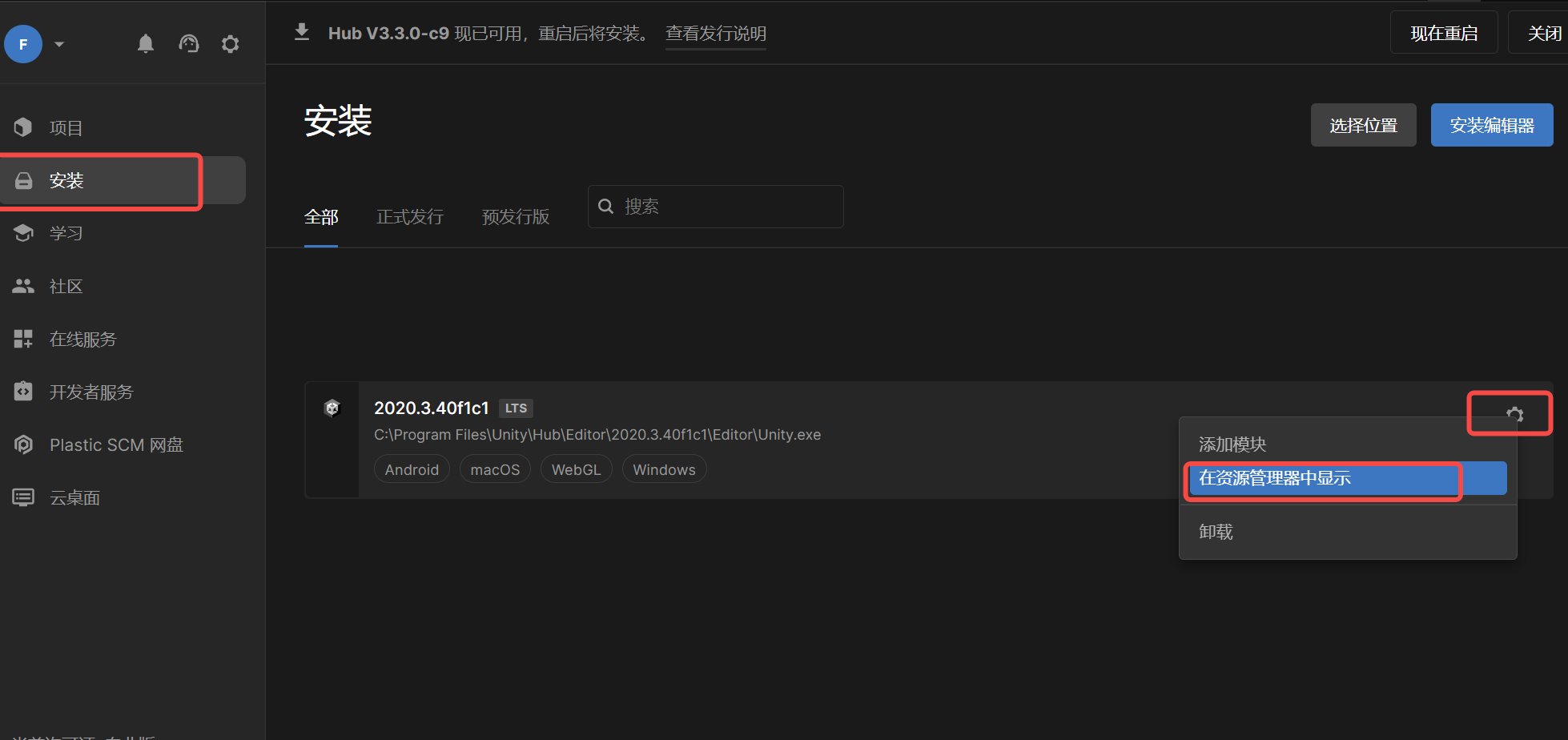
### 1、更改unity默认字体

将更换的字体放入Resouses/Fonts文件夹中

找到unity编辑器默认字体路径

如：C:\ProgramFiles\Unity\Hub\Editor\2020.3.40f1c1\Editor\Data\Resources\PackageManager\BuiltInPackages\com.unity.ugui\Runtime\UI\Core\Text.cs

此路径为unity的安装路径，也可以在unityhub中点击资源管理器中显示查看路径，找到Text.cs文件并打开。



在Text.cs中找到AssignDefaultFont()方法

将

font = Resources.GetBuiltinResource<Font>("Arial.ttf");

改为

font = Resources.Load<Font>("Fonts/AlimamaFangYuanTiVF-Thin");// 为字体文件名，根据实际字体进行修改即可

if (!font)

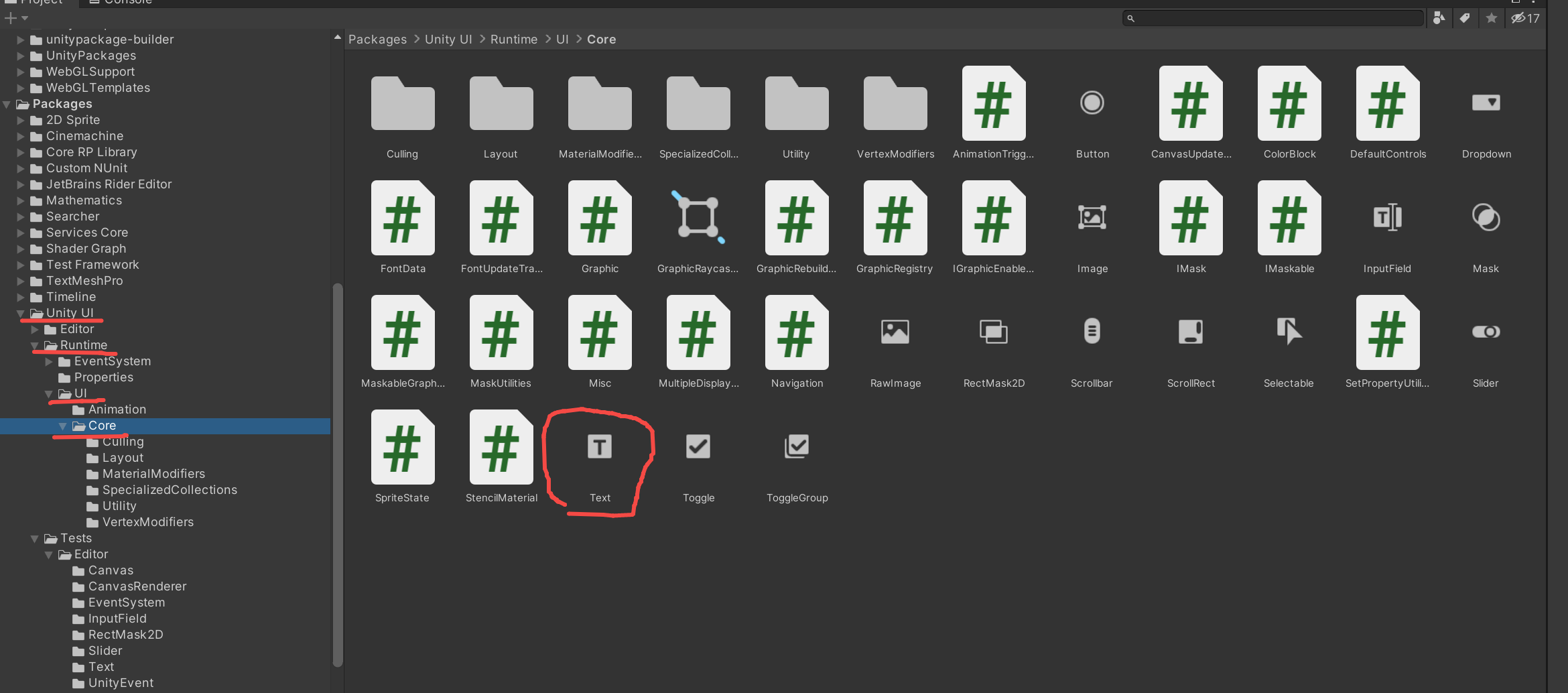
{

font = Resources.GetBuiltinResource<Font>("Arial.ttf");

}

注意，改此文本用vs更改不成功，建议用NotePad++编辑器更改

如果此时在unity中新建text UI控件不是设置的字体，直接在项目的Packag中进行同样的修改（操作如上）



提示：此设置只会对新建的text组件生效，如需对之前做好的text组件字体更改，请结合【[一键换字体工具](https://banding.feishu.cn/wiki/EYnEwW5RniHyDdksf1icauOQnag" \l "FgYfdmdX6oM0jaxuIegcqo1Gnjf)】使用

## WebSocket长链接使用

普通请求举例：

//第一种请求响应方式 适合大系统

private int ReqCode = 0;

public void regist()

{

//MessageId.Types.Enum.ChangeResp是proto生成的，服务器维护消息值，

//客户端直接使用字段，无需关系协议号

WebSocketAgent.AddProxyMsg((uint)MessageId.Types.Enum.ChangeResp, OnChangeSkinResp);

//模拟同一个响应多个地方注册监听 或响应需要不同proto结构反序列化

WebSocketAgent.AddProxyMsg((uint)MessageId.Types.Enum.ChangeResp, OnChangeSkinResp1);

}

public void unRegist()

{

WebSocketAgent.DelProxyMsg((uint)MessageId.Types.Enum.ChangeResp, OnChangeSkinResp);

}

private void OnChangeSkinResp(uint clientCode, ByteString data)

{

//反序列化

ChangeSkinResp resp = ChangeSkinResp.Parser.ParseFrom(data);

... ...

}

private void OnChangeSkinResp1(uint clientCode, ByteString data)

{

//clientCode就是WebSocketAgent.SendMsg返回值，每次请求都会是个新值

//如果同一个响应注册了多个监听 可根据clientCode区分

if (clientCode == ReqCode)

{

//反序列化

**ChangeSkinResp1** resp = **ChangeSkinResp1**.Parser.ParseFrom(data);

... ...

}

}

public void SendChangeSkinReq()

{

//序列化过程封装在WebSocketAgent中，发送是不需关心序列化

ChangeSkinReq changeSkin = new ChangeSkinReq();

changeSkin.SkinId = 0;

changeSkin.UserId = 0;

//如果同一个响应注册了多个监听 存一下ReqCode

ReqCode = WebSocketAgent.SendMsg((uint)MessageId.Types.Enum.ChangeReq, changeSkin);

//如果同一个响应只有自己使用，就注册了一个监听，则不需要记录ReqCode，

//在响应中无需对比clientCode

WebSocketAgent.SendMsg((uint)MessageId.Types.Enum.ChangeReq, changeSkin);

}

//第二种请求响应方式 适合小业务

//类似http请求带回调的发送方式

public void SendChangeSkinReq()

{

//序列化过程封装在WebSocketAgent中，发送是不需关心序列化

ChangeSkinReq changeSkin = new ChangeSkinReq();

changeSkin.SkinId = 0;

changeSkin.UserId = 0;

WebSocketAgent.SendMsg((uint)MessageId.Types.Enum.ChangeReq, changeSkin,

(code, bytes) => {

//code 是错误码 0:成功 不再是ClientCode

if(code == 0)

{

ChangeSkinResp resp = ChangeSkinResp.Parser.ParseFrom(bytes);

}

});

}

**普通请求**有断线缓存机制，断线重连之后会立即重发。

**普通请求**使用的事件广播机制，添加对应协议码的监听回调即可。

高频同步使用IWebNetView进行广播，直接推送解析，省去事件轮询广播，增加效率。

详细代码看WebSocketAgent.cs

高频率同步示例：

//第三种请求响应方式 适合高频率同步 和 模块化业务

public interface IWebNetView

{

void BindNetAgent(WebSocketAgent agent);

void UnBindNetAgent();

bool Contains(uint proxy);

void Push(uint proxy, ByteString dataBytes);

void OnMsgError(uint proxy, int errorCode);

void OnNetRestore();

}

//房间同步

public class RoomSyncNetView : BaseNetView

{

private WebSocketAgent agent;

private List<uint> syncProxy = new List<uint>();

public override void DoAddProxy()

{

base.DoAddProxy();

syncProxy.Add((uint)MessageId.Types.Enum.MoveRes);

syncProxy.Add((uint)MessageId.Types.Enum.EnterRoom);

}

public override void BindNetAgent(WebSocketAgent agent)

{

this.agent = agent;

this.agent.NetView = this;

}

public override void UnBindNetAgent()

{

agent.NetView = null;

agent = null;

}

public override bool Contains(uint proxy)

{

return syncProxy.Contains(proxy);

}

public override void Push(uint proxy, ByteString dataBytes)

{

MessageId.Types.Enum msgId = (MessageId.Types.Enum)proxy;

switch (msgId)

{

//同步房间内所有玩家的移动

case MessageId.Types.Enum.MoveRes:

Move move = Move.Parser.ParseFrom(dataBytes);

...

...

break;

//玩家有进入房间

case MessageId.Types.Enum.EnterRoom:

...

...

break;

}

}

//当前网络连接断开了回调

public override void OnMsgError(uint proxy, int errorCode)

{

//同步出错了 判断当前是否是断网了

//断网了就暂停每秒N次的同步

if (!agent.IsConnected)

{

//暂停同步

}

}

//当前网络恢复了回调

public override void OnNetRestore()

{

//恢复同步

}

}

//世界同步

public class WorldSyncNetView : BaseNetView

{

...

...

public override void Push(uint proxy, ByteString dataBytes)

{

MessageId.Types.Enum msgId = (MessageId.Types.Enum)proxy;

switch (msgId)

{

//例如同步世界中的资源刷新

case MessageId.Types.Enum.Assets:

Assets asset = Assets.Parser.ParseFrom(dataBytes);

...

...

break;

}

}

...

...

}

//使用方式示例

public class Main

{

private RoomSyncNetView RoomSyncNet;

private WorldSyncNetView WorldSyncNet;

public Main()

{

//打开房间同步

RoomSyncNet = new RoomSyncNetView();

RoomSyncNet.BindNetAgent(WebSocketAgent.Ins);

//关闭房间同步

RoomSyncNet.UnBindNetAgent()

//打开世界同步

WorldSyncNet = new WorldSyncNetView();

WorldSyncNet.BindNetAgent(WebSocketAgent.Ins);

//关闭世界同步

WorldSyncNet.UnBindNetAgent()

}

}

//基于第三种模块化方式 支持小模块

public class SmallSyncNetView : BaseNetView

{

public override void DoAddProxy()

{

base.DoAddProxy();

syncProxy.Add((uint)MessageId.Types.Enum.xxxxxxx);

}

public override void BindNetAgent(WebSocketAgent agent)

{

this.agent = agent;

this.agent.CurProcess.RegisterNetView(this);

}

public override void UnBindNetAgent()

{

this.agent.CurProcess.UnRegisterNetView(this);

this.agent = null;

}

public override void Push(uint proxy, ByteString dataBytes)

{

MessageId.Types.Enum msgId = (MessageId.Types.Enum)proxy;

switch (msgId)

{

case MessageId.Types.Enum.xxxx:

Assets asset = Assets.Parser.ParseFrom(dataBytes);

...

...

break;

}

}

public void Open()

{

UnBindNetAgent();

BindNetAgent(WebSocketAgent.Ins);

}

public void Close()

{

UnBindNetAgent();

}

}

public class Main

{

private SmallSyncNetView SmallSyncNetView;

public Main()

{

SmallSyncNetView= new SmallSyncNetView();

//注册

SmallSyncNetView.BindNetAgent(WebsocketAgent.Ins);

//解注

SmallSyncNetView.UnBindNetAgent();

//从Agent中获取

SmallSyncNetView next = WebsocketAgent.Ins.CurProcess.GetNetView<SmallSyncNewView>();

//websocket注册子NetView, NetView内部无需添加this.agent.CurProcess.RegisterNetView(this);

WebsocketAgent.RegisterSubNetView(SmallSyncNewView);

WebsocketAgent.UnRegisterSubNetView(SmallSyncNewView);

SmallSyncNetView netView = WebsocketAgent.GetSubNetView<SmallSyncNetView>();

}

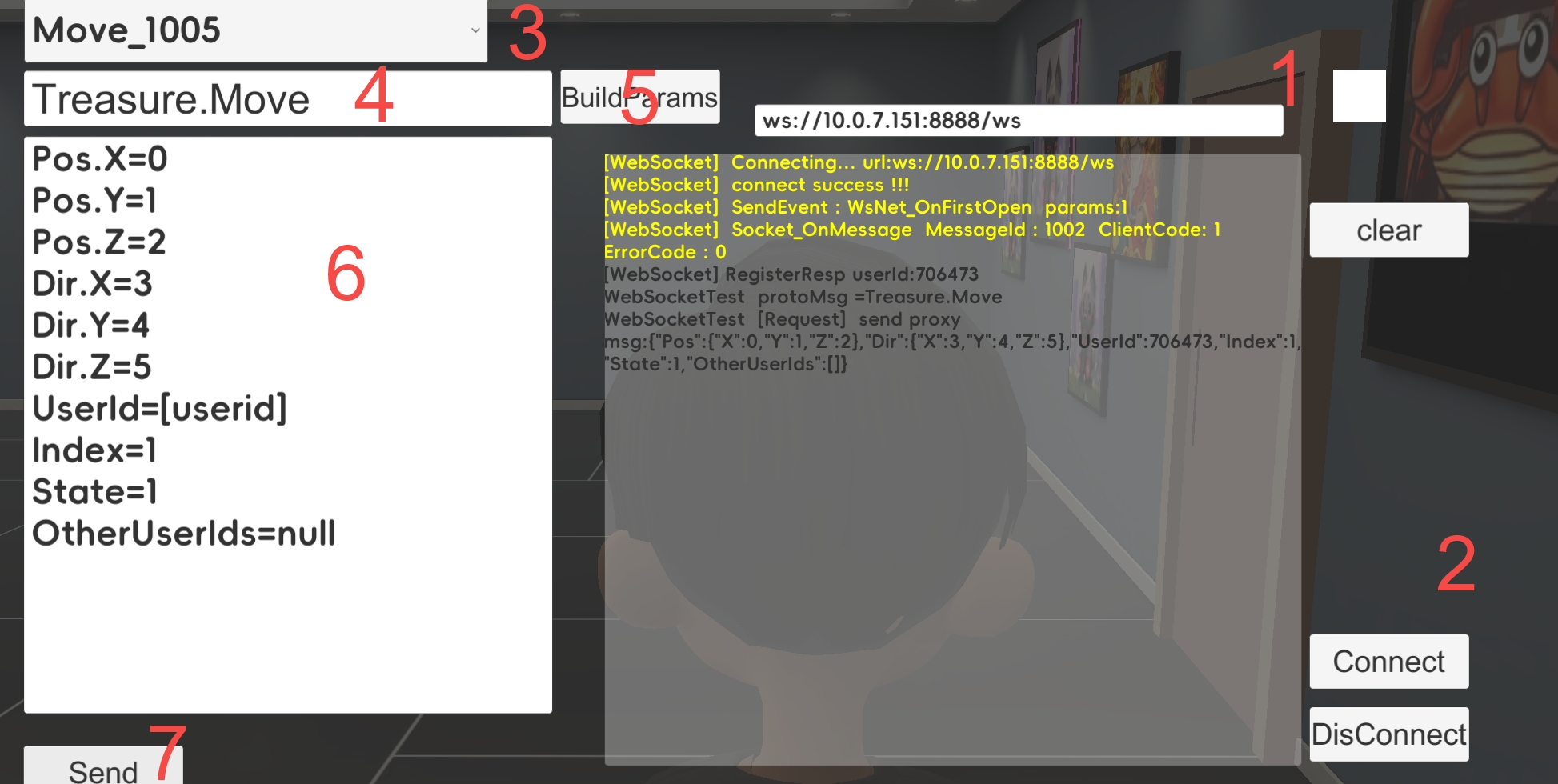
}

## WebSocket长链接协议测试工具

工具入口：UnityToolbar/Tools/ProxyTestTool

在Unity中的Hieratchy视图中会生成一个[ProxyTestTool]的GameObject

进入游戏在屏幕上快速点击 1.5s内至少点击6次，弹出工具页面



确认ws的链接地址

点击Connect log中显示ConnectSuccess 说明链接成功

选择要测试的协议

确认要发送的protocol协议结构,不是生成的协议名，需自己手动更改

点击BuildParams 生成参数列表6

编辑参数 参数带有[]的 如：[userid]会自动填充玩家的userid 目前支持[userid][token][gametoken][landid][sceneid][roomid]

发送

## WebSocket定制化

因后续可能大量内容的开发，为了满足足够多的场景应用，现将websocket进行定制化抽象

Websocket长链接有WebSocketAgent, WebSocketConfig, WebSocketProcess三部分组成

WebSocketAgent：负责链接，收发消息

WebSocketConfig：负责对WebSocketAgent进行配置，如：链接地址，协议类型，心跳，断线重连等

WebSocketProcess：负责对WebSocketAgent收到的消息进行处理，是具体的实现。

使用方法：

//假如需求是希望使用Websocket传json 想要收发json字符串传递数据

//继承WebSocketConfig

public class NewWebSocketConfig : WebSocketConfig

{

public NewWebSocketConfig()

{

Address = "XXX/XXX/X/";

UseByte = false;

UseText = true;

HeartBeatEnable = false;

... ...

}

public override WebSocketProcess GetProcess(WebSocketAgent agent)

{

... ...

return new WebSocketTextProcess(agent);

}

}

//json字符串协议的处理过程

public class WebSocketTextProcess: WebSocketProcess

{

public WebSocketTextProcess(WebSocketAgent agent) : base(agent)

{

}

public override void OnFristOpen(OpenEventArgs e)

{

//第一次链接成功

}

public override void OnReconnectOpen(OpenEventArgs e)

{

//重连成功

}

public override void OnSocket\_OnMessageSuccess(string data)

{

//收到json字符串 处理过程

... ...

}

public override void Socket\_OnClose(CloseEventArgs e)

{

//链接关闭

}

public override void Socket\_OnError(ErrorEventArgs e)

{

//链接报错了

}

}

//测试连接

public class Main

{

public Main()

{

//启动websocket链接

WebSocketAgent.Ins.Connect(new NewWebSocketConfig());

}

}

# TA

## webgl图形

渲染

WebGL2.0 Unity WebGL 仅支持延迟渲染路径。在 WebGL1.0 上，Unity WebGL 运行时将回退到**前向渲染**。

全局光照

仅支持**烘焙 GI**（Global illumination)。不支持实时 GI。此外，仅支持**非方向**光照贴图。

着色器限制

  仅允许使用常量表达式、循环索引或组合的方式进行动态索引。唯一的例外是顶点着色器中的 uniform 访问，这种情况可使用任何表达式进行索引。

  允许的唯一循环类型是计算 for 循环，在这种情况下，初始化函数 (initializer) 将变量初始化为常量，更新 (update) 向变量添加常量或从变量中减去常量，而延续测试 (continuation test) 会将变量与常量进行比较。不允许使用不符合这些条件的 for 循环以及 while 循环。

  着色器目前仅支持**target 3.0**。

抗锯齿

相机中开启的**后处理效果**、**HDR**都与抗锯齿**不兼容**

不同的多重采样级别没有影响

反射探针

  支持反射探针，但由于 WebGL 规范中存在有关渲染到特定 Mipmap 的限制，因此不支持平滑实时反射探针（因此实时反射探针将始终产生锐利的反射，这种情况下可能看起来分辨率非常低）

更多细节见unity[官方文档](https://docs.unity.cn/cn/2021.1/Manual/webgl-graphics.html)。

## 静态资源准入

### ****模型****

DCC**导出设置**

尽可能使用fbx格式

与引擎统一单位

导出网格必须是多边形拓扑，不能是贝塞尔曲线，样条曲线、细分曲面等

确保所有deformers都烘培到网格模型上：如骨骼的形变已经烘焙到蒙皮的权重上等

不建议模型中使用的纹理随模型导出

如果需要导入blend shape normals 必须指定光滑组smooth groups

导出时不要携带摄像机灯光材质等场景信息

**影响性能关键因素**

最小的面数 不需要微三角面 三角面尽量分布均匀

合理化拓扑结构与平滑诅，尽可能是闭包，否则可能产生烘焙错误

尽量少的材质个数

尽可能少的蒙皮网格

尽可能少的骨骼数量

fk与ik节点分离 导出时删除ik骨骼节点

optimize mesh data启用 会根据材质删除不需要的数据（**如果模型中额外写入了除UV1,UV2的UV信息，或者顶点色等信息，着色器用到时需要取消勾选此选项，否则打包时会自动剔除这些信息使其丢失**）

### ****动画****

skin weights 默认四根骨骼 不重要的动画对象可以减少至一根 节省计算量

optimize bones 建议开启，在导入时自动剔除一些没有蒙皮顶点的动画

optimze game object 动画文件无论压缩与否都会与ddc软件中表现有所差异

compression

keyframe reduction 减少冗余关键帧，减少动画文件内存

keyframe reduction and commpression 减少关键帧的同时对关键帧储存数据进行压缩，只影响文件大小

optimal 仅适用于generic 和 humanoid动画类型 unity决定如何进行压缩

**优化原则**

效果差异

曲线效果（总曲线与各种曲线数量，常量曲线比重大更好）

动画文件大小 动画文件在小几百k或更少合理，超过1m需要考虑是否合理

# 项目管理

## 一、分支管理

### 1、分支介绍

master：主分支，用于存放最稳定的正式分支

develop:开发分支，用于日常开发，存放最新的开发版

feature\_"个人名称"：个人分支，用于个人提交本身的修改

hotfix\_"补丁版本号"：补丁分支，用于修复正式版本分支。

### 2、分支使用

feature\_"个人名称"，每个开发人员都需要从develop分支下，新建一个 个人分支，用于自己的日常开发及向develop合并。个人提交内容，直接提交到自己个人分支，由项目管理人员管理合并到develop分支。

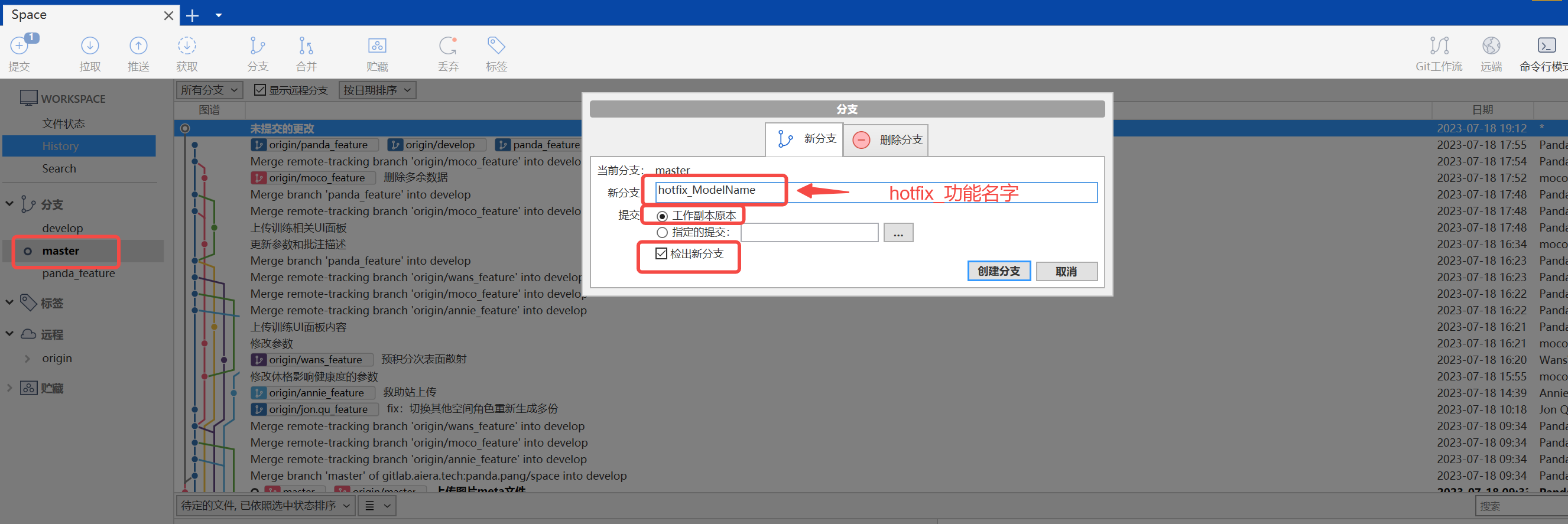
develop分支:开发人员日常开发与功能测试发布用的总分支，里面为最新的开发内容，个人分支获取最新内容，直接拉取develop分支到自己个人分支。

master分支:为正式稳定的分支，在develop分支或者hotfix补丁分支测试无误后，统一合并到master分支，用于存放正式稳定的版本。

hotfix\_补丁分支，用于修复正式版本，临时性分支。从master分支新建一个补丁分支，开发完成后，合并入master分支中，并同步到develop分支。然后删除。

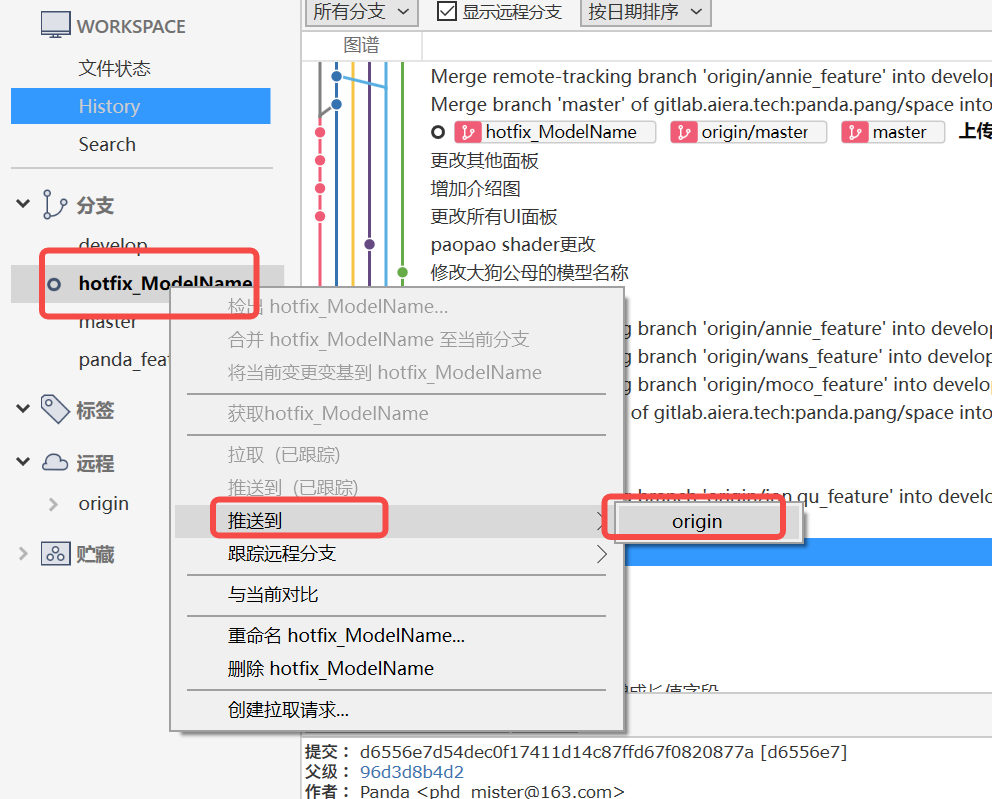
#### 1、 hotfix\_补丁分支具体使用操作

在master分支的基础上检出一个新的分支，命名以hotfix\_功能名。

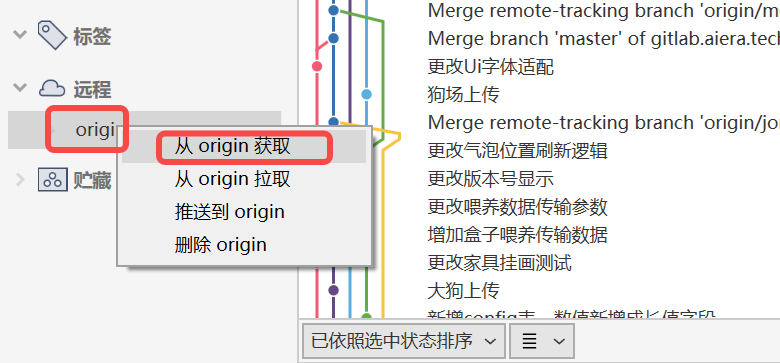


是否多人协作，如果是多人协作，需要将hotfix分支推送到origin。 如果只需要个人修改，则不需要推送到origin。（个人修改，就到此为止，修复完毕后，将此分支合并到master中，然后删除此分支）

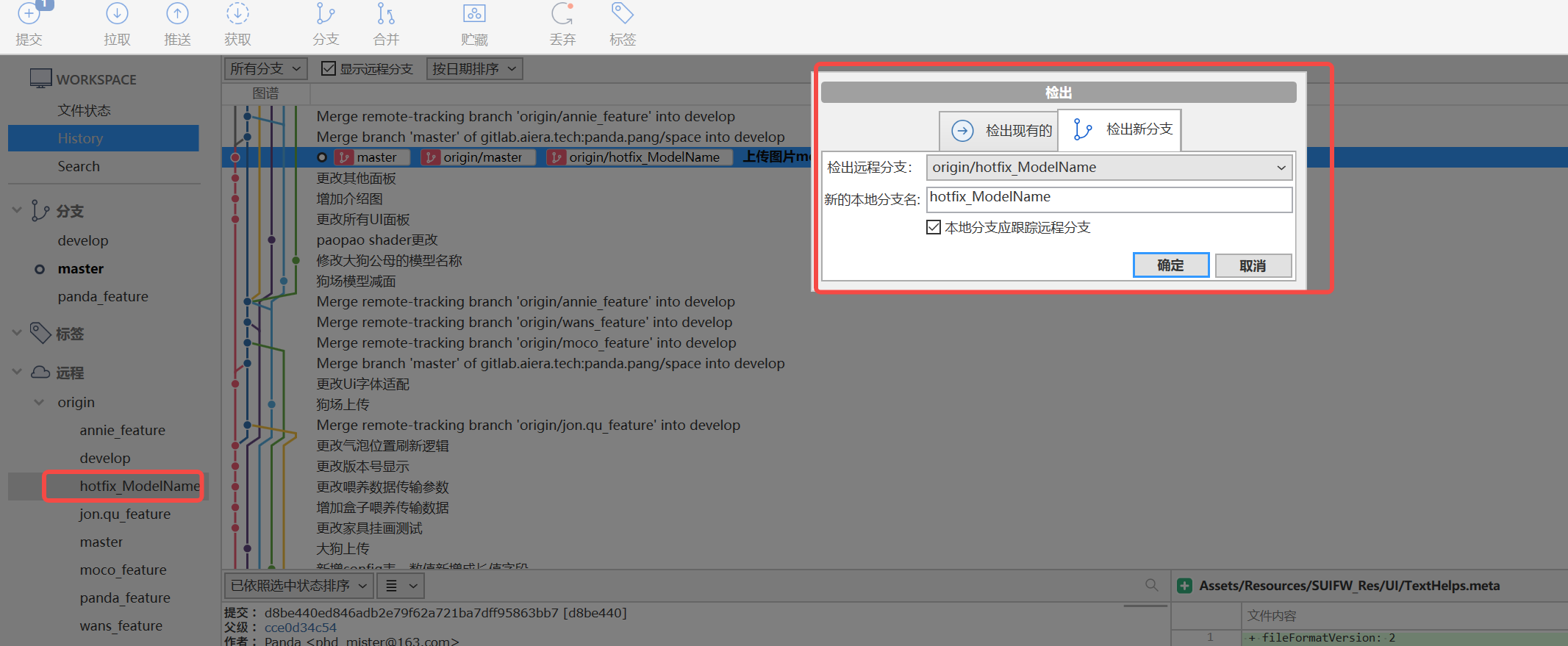
推送到origin



在远程--> orgin 中，查看是否存在hotfix分支。如果不存在就重新从orgin拉取一遍



从orgin中选中hotfix分支，双击检出新分支，就可以切换到此分支进行拉取、合并、开发



最后在此分支上开发完毕后，由负责该版本的开发人员，将hotfix分支合并到master分支上，然后把origin中的fotfix分支删除。所有人员将自己本地检出的fotfix分支也都删除掉。

### 3、提交内容规范

每个人要维护自己个人分支与develop分支的同步性。

每次更改本地内容前，都先拉取develop分支，确保自己个人分支与develop分支内容同步。

每次拉取develop前，要先检查个人分支里面的内容有没有修改或者新增，如果是需要提交的修改和新增的文件，先提交到个人分支，然后把无关的文件内容都删除和还原。

如果在拉取develop前，本地没有需要提交的，那就把个人分支里面有变动的内容都还原，然后再拉取develop分支。并把新内容推送到个人分支。

如果在拉取develop前，本地有提交的，提交上去后，等待或者通知管理员，把自己的分支内容合并到develop后，然后自己再重新拉取develop分支的内容。

大家维护个人分支，只需要记住 在本地无修改时，先拉取develop分支，让自己分支同步。本地有修改，优先处理个人分支，该提交的提交，该还原的还原。

如果你本次提交了A文件，develop分支还没有合并，但是你还要修改A文件，此时有两种选择， 1，那你就把A文件修改之后，再次上传，不拉取develop。 2，通知管理员，把自己的分支合并。然后拉取下来之后再进行修改。

每次提交都只提交自己修改的文件，无论是什么情况。 （就算解决完冲突之后，也要查看暂存区是不是自己修改的文件），不是自己要提交的文件，都要及时处理还原和删除，不要提交上去。

核心点在于，1，要及时保持自己个人分支与develop的同步性，2，只提交自己要提交的东西，不是自己要提交的东西，一律还原或者删除。

Git每次提交内容，都要写commit message（提交说明），提交说明内容要求：清晰明了，说明本次修改、新增、删除等变更的内容、本次提交的目的。

# 版本管理

# 代码管理

## 各个管理器

### 1.CharacterManager

characterManager负责动态加载角色，便于加载不同场景的角色管理。

public void Load()

{

int roleId = PlayerPrefs.GetInt("CurCharacterId");

m\_LastCharacterId = roleId;

if (roleId > 0)

{

hotman m\_CurHotManConfig = ManageMentClass.DataManagerClass.GetHotmanTableFun(roleId);

string m\_CurHotManName = m\_CurHotManConfig.hotman\_mode;

string path = string.Format("{0}{1}", SysDefine.SYS\_PATH\_CHARACTER, m\_CurHotManName);

GameObject m\_CurHotManObj = ResourcesMgr.GetInstance().LoadAsset(path, true);

playerItem = m\_CurHotManObj.GetComponent<PlayerItem>();

if (playerItem == null)

{

playerItem = m\_CurHotManObj.AddComponent<PlayerItem>();

}

....

}

...

}

Load接口动态根据设置roleId,加载角色。

public void ChangeRole(int roleId)

{

if (m\_LastCharacterId == roleId)

return;

hotman m\_LastHotManConfig = ManageMentClass.DataManagerClass.GetHotmanTableFun(m\_LastCharacterId);

hotman m\_CurHotManConfig = ManageMentClass.DataManagerClass.GetHotmanTableFun(roleId);

string m\_CurHotManName = "";

string m\_LastHotManName = "";

if (m\_CurHotManConfig != null && m\_LastHotManConfig != null)

{

...

}

...

}

ChangeRole接口根据roleId切换角色。

CharacterManager里面还有一些设置角色位置旋转，跟随相机相关的接口，具体实现看代码。

### 2.PlayerCtrlManager

主要控制角色移动相关，实例化的节点下面会挂载**MoveController，MoveController**里实现角色移动的逻辑**。**

初始化接口**Init**会设置moveController的摇杆和主相机**。**

public void Init()

{

...

JoyStick joyStick = JoystickManager.Instance().GetJoyStick();

if (joyStick != null)

{

moveController.moveControllJoyStick = joyStick;

}

CameraRotateController cameraRotateController = JoystickManager.Instance().GetCameraRotateController();

if (cameraRotateController != null)

{

moveController.cameraRotateController = cameraRotateController;

}

moveController.MinaCamera = Camera.main;

...

}

里面还有一些设置触摸事件的接口，主要用于跳转新场景后重新绑定。

### 3.CameraManager

主要用来动态加载CinemachineBrain CinemachineFreeLook，设置角色头顶跟随相机。

### 4.JoystickManager

控制摇杆，动态加载摇杆。

Editor模式下用EasyTouch控制角色移动，真机模式挂载Joystick通过触摸事件处理移动操作；

public void ReadTouch(TouchState touch, Vector3 positionA)

{

if (!transform.gameObject.activeSelf)

{

return;

}

if (touch == TouchState.down)

{

//触发的手指按下的事件

// circle.position = touch.position;

onDown.Invoke(circle.position, positionA, maxOffset);

}

else if (touch == TouchState.move)

{

//记录之前的位置

Vector3 prePointPos = point.position;

//记录偏移量

Vector3 offset = new Vector3(positionA.x, positionA.y, 0) - prePointPos;

//小圈跟着一起移动

point.position += offset;

//限制摇杆的移动

if ((point.position - circle.position).magnitude > maxOffset)

{

point.position = circle.position + (point.position - circle.position).normalized \* maxOffset;

}

//计算归一化偏移量

xAxis = offset.x / maxOffset;

yAxis = offset.y / maxOffset;

onDrag.Invoke(circle.position, point.position, maxOffset);

}

else if (touch == TouchState.up)

{

//手指抬起

//重置小圈的位置到大圈的中心

point.position = circle.position;

//触发手指抬起事件

onUp.Invoke(circle.position, point.position, maxOffset);

}

}

### 5.TransferEffectManager

主要用来处理跳转场景的传送台，进入传送台触发传送加载传送特效等。

### 6.RTManager

主要用来在UI面板预览3D模型，主要用RenderTexture来实现,将渲染3D物体的相机渲染的RT赋给一个RawImage。



LoadRTMode()接口用来预览场景中的家具

public void LoadRTModel(int itemId)

{

...

item m\_ItemConfig = ManageMentClass.DataManagerClass.GetItemTableFun(itemId);

if (m\_ItemConfig != null)

{

...

}

...

}

LoadCharacter()接口用来预览角色

public void LoadCharacter()

{

Transform modelParent = UIManager.GetInstance().GetModelTrans();

if (modelParent != null)

{

...

modelParent.transform.localPosition = new Vector3(2000f, 0f, 0f);

modelParent.transform.localRotation = Quaternion.Euler(Vector3.zero);

}

}

### 7.SceneLoadManager

用于加载场景管理，加载不同场景处理不同逻辑。

这里遇到一个坑点：WebGL平台异步加载**SceneManager.LoadSceneAsync**算进度条逻辑不能写到协程里，

要写到FixedUpdate里不然真机跑会卡Loading。

### 总结：

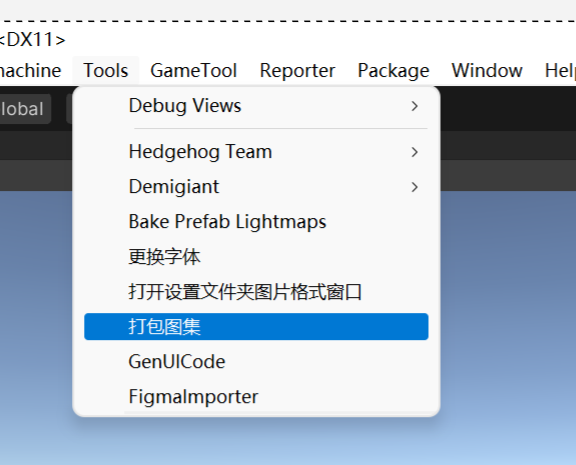
其实开发这么多管理器就是为了代码看着更规范，各个模块之间降低耦合度，清晰明了，便于扩展。

### 性能相关：TODO

#### 图集整理

目前主要着手将UI散图打成图集，降低DrawCall。但目前还没有规范性的UI切图标准及整理那些通用的切图到一张图集里去，像大的切图其实可以考虑直接用原Texture不打成图集。目前只整理了一部分。

目前做了一个自动化打图集工具：



执行这个会将"Assets\UISprite的Sprite打成图集放到Assets\Resources\UIRes\Atlas目录下，同时把原资源目录从Resources目录下移出去了，能减小包体大小。

#### 打表工具优化

之前打表工具读取Excel配置时，实际行数与表格真实行数有出入（有时会大于真实行数），导致读取时总是越界报错。手动删除表格末尾空白行也能解决问题，但不保障策划每次都有这种操作。

代码在生成Excel二进制数据时做了保护:计数空白列与总列数一致及判定该行是空白行

private static void GenerateExcelBinary(DataTable table)

{

...

int columnNullNum = 0;

for (int j = 0; j < table.Columns.Count; j++)

{

if (row.IsNull(j))

{

columnNullNum++;

}

}

if (columnNullNum == table.Columns.Count)//大于实际行不做处理

{

break;

}

...

}

# PC适配

## 窗口兼容

window.onresize = function () {

var widthScale=window.innerWidth/1920;

var heightScale=window.innerHeight/1080;

var averageScale=1;

if(widthScale>heightScale)

{

averageScale=heightScale;

}

else

{

averageScale=widthScale;

}

canvas.style.width = (1920\*averageScale) + "px";

canvas.style.height = (1080\*averageScale)+ "px";

document.getElementById("custom-image").style.width =

(1920\*averageScale) + "px";

document.getElementById("custom-image").style.height =(1080\*averageScale)+ "px";

}

1，算出设备屏幕与1920\*1080中的比例系数，做一个最大尺寸填充的尺寸。这样可以使unity元素不被拉伸。

2，window.onresize,此方法在更改窗口时会被调用。

3，上述方法要写在 index.html 配置中。

## 移动端PC端同时兼容

1，要在index配置文件中，对窗口大小的设置，通过下面判断识别，分开设置窗口适配

if (/iPhone|iPad|iPod|Android/i.test(navigator.userAgent)) {

container.className = "unity-mobile";

// Avoid draining fillrate performance on mobile devices,

// and default/override low DPI mode on mobile browsers.

config.devicePixelRatio = 1;

// unityShowBanner('WebGL builds are not supported on mobile devices.');

}

else

{

//pc

}

# 全屏设置

自定义按钮监听 UnityInstanc.SetFullscreen(1);，点击按钮全屏显示。此方法 必须手动调用， 不支持代码直接运行调用

script.onload = () => {

createUnityInstance(canvas, config, (progress) => {

progressBarFull.style.width = 100 \* progress + "%";

}).then((unityInstance) => {

loadingBar.style.display = "none";

customImage.style.display = "none";

fullscreenButton.onclick = () => {

unityInstance.SetFullscreen(1);

};

}).catch((message) => {

alert(message);

});

};

# 运行平台识别

GetOperationPlatformFun:function()

{

var ua = navigator.userAgent.toLowerCase();

var ipad= ua.match(/ipad/i) == "ipad";

var iphone= ua.match(/iphone os/i) == "iphone os";

var midp= ua.match(/midp/i) == "midp";

var uc7= ua.match(/rv:1.2.3.4/i) == "rv:1.2.3.4";

var uc= ua.match(/ucweb/i) == "ucweb";

var android= ua.match(/android/i) == "android";

var windowsce= ua.match(/windows ce/i) == "windows ce";

var windowsmd=ua.match(/windows mobile/i) == "windows mobile";

if (!(ipad || iphone || midp || uc7 || uc || android || windowsce || windowsmd)) {

// PC 端 返回1

return 1;

}else{

// 移动端 返回2

return 2;

}

}GetOperationPlatformFun:function()

{

var ua = navigator.userAgent.toLowerCase();

var ipad= ua.match(/ipad/i) == "ipad";

var iphone= ua.match(/iphone os/i) == "iphone os";

var midp= ua.match(/midp/i) == "midp";

var uc7= ua.match(/rv:1.2.3.4/i) == "rv:1.2.3.4";

var uc= ua.match(/ucweb/i) == "ucweb";

var android= ua.match(/android/i) == "android";

var windowsce= ua.match(/windows ce/i) == "windows ce";

var windowsmd=ua.match(/windows mobile/i) == "windows mobile";

if (!(ipad || iphone || midp || uc7 || uc || android || windowsce || windowsmd)) {

// PC 端 返回1

return 1;

}else{

// 移动端 返回2

return 2;

}

}

1，此方法是通过配置在jslib文件中。被c#方法调用拿取，在打包成web端是可以拿取具体平台。

2，通过 # if UNITY\_ANDROID 方式在web端判断不出，通过 RuntimePlatform platform = Application.platform; 方式同样也拿不到。

## 多平台多入口判断识别

运行平台的识别可以通过GetOperationPlatformFun（） 方法获取

app端进入，还是网页端进入，通过 网址链接后的配置“#p” 识别

# 加密解密

加密解密建议多种规则组合使用，或者增加自定义规则

多规则组合使用如 ，先对内容编码，然后对编码后的内容再进行加密

增加自定义规则，如先对内容编码或加密。然后再编码对编码和加密的内容中，增加自定义的内容，然后再加密或编码。 解密的时候，可以根据自定义的规则去进行处理

## Ase 加密解密

加密和解密都需要一个 key值，和IV值。 加密和解密所需的key值和IV 值要保持一致

Key值要求为 32 位 ，如：12345678976543218765432102345678

IV 值要求位 16 位， 如： 1234789119874321

### Aes加密

/// <summary>

/// Aes 加密

/// </summary>

/// <param name="plainText"></param>

/// <param name="StrKey"></param>

/// <param name="StrIv"></param>

/// <returns></returns>

public static string EncryptString(string plainText, string StrKey, string StrIv)

{

byte[] key = Encoding.UTF8.GetBytes(StrKey);

byte[] iv = Encoding.UTF8.GetBytes(StrIv);

using (AesCryptoServiceProvider aes = new AesCryptoServiceProvider())

{

aes.Key = key;

aes.IV = iv;

ICryptoTransform encryptor = aes.CreateEncryptor(aes.Key, aes.IV);

byte[] encryptedBytes = null;

using (var ms = new System.IO.MemoryStream())

{

using (var cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write))

{

byte[] plainBytes = Encoding.UTF8.GetBytes(plainText);

cs.Write(plainBytes, 0, plainBytes.Length);

}

encryptedBytes = ms.ToArray();

}

return Convert.ToBase64String(encryptedBytes);

}

}

### Aes解密

/// <summary>

/// Aes解密

/// </summary>

/// <param name="encryptedText"></param>

/// <param name="StrKey"></param>

/// <param name="StrIv"></param>

/// <returns></returns>

public static string DecryptString(string encryptedText, string StrKey, string StrIv)

{

byte[] key = Encoding.UTF8.GetBytes(StrKey);

byte[] iv = Encoding.UTF8.GetBytes(StrIv);

using (AesCryptoServiceProvider aes = new AesCryptoServiceProvider())

{

aes.Key = key;

aes.IV = iv;

ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key, aes.IV);

byte[] encryptedBytes = Convert.FromBase64String(encryptedText);

byte[] decryptedBytes = null;

Debug.Log("encry:: " + encryptedBytes.Length);

using (var ms = new System.IO.MemoryStream(encryptedBytes))

{

using (var cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Read))

{

byte[] buffer = new byte[encryptedBytes.Length];

int bytesRead = cs.Read(buffer, 0, buffer.Length);

decryptedBytes = new byte[bytesRead];

Array.Copy(buffer, decryptedBytes, bytesRead);

}

}

return Encoding.UTF8.GetString(decryptedBytes);

}

}

## Base64加码解码

### 编码Base64

// 将字符串编码为 Base64

public static string Base64Encode(this string text)

{

return (Convert.ToBase64String(Encoding.Default.GetBytes(text)));

}

### 解码Base64

// 将 Base64 解码还原字符串

public static string Base64\_Decode(string plainText)

{

// var plainTextBytes = System.Convert.FromBase64String(plainText);

return (Encoding.Default.GetString(Convert.FromBase64String(plainText)));

}

# 不同平台的人物操控

移动平台，主要通过屏幕Touch点去进行识别操控

PC平台，主要通过键盘按键和鼠标按键去进行识别操控

## 移动

1，移动设备平台，通过摇杆控制人物移动

2，pc平台，通过WASD移动

## 镜头旋转控制

1，移动设备平台，通过手指点击滑动

2，pc平台，通过鼠标左键按下移动控制