# 综合

## 克隆

Instantiate

e.g.

**Instantiate**(prefab, new [Vector3](file:///D:\Program%20Files%20(x86)\Unity\Editor\Data\Documentation\html\en\ScriptReference\Vector3.html)(i \* 2.0F, 0, 0), [Quaternion.identity](file:///D:\Program%20Files%20(x86)\Unity\Editor\Data\Documentation\html\en\ScriptReference\Quaternion-identity.html)) as [Transform](file:///D:\Program%20Files%20(x86)\Unity\Editor\Data\Documentation\html\en\ScriptReference\Transform.html);

this.highLight = (Material) Object.**Instantiate**(MBattle.MaterialHL);

## asset bundle

## Resources.Load

eActor.shadow = Object.Instantiate(Resources.Load("Shadow/OtherShadow")) as GameObject;

## U3D 模型制作规范：

http://blog.csdn.net/tammy520/article/details/8701389

## 常用插件

一 界面制作 推荐：NGUI  
二 2D游戏制作 推荐：2D Toolkit  
三 可视化编程 推荐：PlayMaker  
四 插值插件 推荐：iTween,HOTween  
五 路径搜寻 推荐：Simple Path  
六 美术及动画制作 推荐：RageSpline,Smooth Moves  
七 画面增强    推荐：Bitmap2Material,Strumpy Shader Editor  
八 摄像机管理  推荐：Security Camera  
九 资源包  推荐：Nature Pack

## Shade在哪里用

## 场景

## 角色动作

## UI贴图

## 场景加载

Application.LoadLevel("mainscene");

## 管理器的存在形式

全局对象(管理器、单件)生成的时候一般 会创建一个**空的GameObject**,然后把脚本(对象、Component)挂在空Object下，比如：

public static NetworkManager GetInstance()

{

if (!Instance)

{

Container = **new GameObject()**;

Container.name = "NetworkManager";

Instance = Container.AddComponent(typeof(NetworkManager)) as NetworkManager;

}

return Instance;

}

全局的对象需要有全局的GameObject与之关联：

public class BulletManager : MonoBehaviour {

public static BulletManager Instance;

public static GameObject Container;

对象生成时机：

在代码中调用生成，一般初始化完成后各个管理器对象已经生成并建立好相应的关联

## 第三方库

using Lidgren.Network;

库代码路径：Unity\Assets\3rdParty\Lidgren.Network

## Prefab

Object PlayerBigPrefab;

Object PlayerBulletPrefab

PlayerShot()

{

GameObject.Instantiate(PlayerBigPrefab, location, rotation \* BulletRotation);

}

void Start ()

{

PlayerBigPrefab = Resources.Load("Bullets/PlayerBigShot");

PlayerBulletPrefab = Resources.Load("Bullets/PlayerBulletPrefab");

}

## 名字空间

CrabBattleServer.CrabBehavior cb;

这里CrabBattleServer是名字空间

## 各种成员初始化方式

**方式1** 隐藏得好深

public class NetworkManager : MonoBehaviour {

public GameObject Enemy;

public CrabManager EnemyManager;

。。。

**Enemy** = GameObject.Instantiate(Resources.Load("battlecrab"),..as GameObject;

Enemy.animation.Play("laying");

**// CrabManager竟然藏在prefab:battlecrab里**

**EnemyManager** = Enemy.GetComponent<CrabManager>();

}

**方式2**

Netman = NetworkManager.GetInstance();

**方式3** 直接加载资源

**//** Object BulletSmallShot;

BulletSmallShot = Resources.Load("Bullets/EnemySmallShot");

**方式4** 克隆

projectile = GameObject.Instantiate(BulletSmallShot,。。。

## 血条在哪里

血条是Healthbar(Prefab)里的一张纹理

healthbarobject = GameObject.Instantiate(Resources.Load("Healthbar")) as GameObject;

healthbar = healthbarobject.GetComponent<GUITexture>();

# Idea

## MMO开副本玩策略单机

好友间可查看分数，由于是单机，分数可造假，好友间看看而已，造假也没什么意义

# Warrying分析

## 消息

**注册：**

**注册消息**

消息分为两种，固定消息和动态消息，固定消息直接在客户端初始化如下：

public static void bindFixedMessage()

{Message.messages["Loginapp\_importClientMessages"] = new Message(5, "importClientMessages", 0, 0, new List<Byte>(), null);

Message.messages["Client\_onImportClientMessages"] = new Message(518, "Client\_onImportClientMessages", -1, -1, new List<Byte>(),

**KBEngineApp.app.GetType().GetMethod("Client\_onImportClientMessages"));**

动态消息由服务器发送给客户端：



上图messages只保存有消息名字的消息 （有消息名必有消息号，反之未必）

clientMessages保存所有带有”Client\_”标志的消息 (键值:消息ID)

loginappMessages保存与登入服务器交互的消息 (键值:消息ID)

baseappMessages保存与其它服务器交互的消息 (键值:消息ID)

**发送消息：**

bundle.send

//请求客户端消息

bundle.newMessage(Message.messages["Loginapp\_importClientMessages"]);

//请求服务器错误描述

bundle.newMessage(Message.messages["Loginapp\_importServerErrorsDescr"]);

// loginapp心跳包

bundle.newMessage(Message.messages["Loginapp\_onClientActiveTick"]);

//发送版本号

bundle.newMessage(Message.messages["Loginapp\_hello"]);

bundle.writeString(clientVersion);

bundle.writeString(clientScriptVersion);

bundle.writeBlob(\_clientdatas);

bundle.send(\_networkInterface);

bundle.newMessage(Message.messages["Loginapp\_onClientActiveTick"]);

//服务器hello返回

《———— 521

//发送用户名和密码

bundle.newMessage(Message.messages["Loginapp\_login"]);

bundle.writeInt8(\_clientType); // clientType

bundle.writeBlob(new byte[0]);

bundle.writeString(username);

bundle.writeString(password);

《———— 502 Client\_OnLoginSuccessfully

login\_baseapp

**currserver = "baseapp";**

**currstate = "";**

bundle.newMessage(Message.messages["Baseapp\_hello"]);

bundle.writeString(clientVersion);

bundle.writeString(clientScriptVersion);

bundle.writeBlob(\_clientdatas);

bundle.newMessage(Message.messages["Baseapp\_importClientMessages"]);

**Event.fireAll("Baseapp\_importClientMessages", new object[]{});**

**接收消息：**

process()

recv();

msgReader.process(\_datas, (MessageLength)successReceiveBytes);

Message msg = Message.clientMessages[**msgid**]; //根据消息ID找到对应的消息及处理函数

msg.handleMessage(stream);

handler.Invoke(KBEngineApp.app, new object[]{msgstream});

…

Client\_onImportClientMessages();**//这是具体调用例子，具体调用哪个函数由消息定义时确定，参考上面的bindFixedMessage**

Ui\_font\_source.txt里的：

<Client::onImportClientMessages>

<id>518</id>

<descr>服¤t务?器¡Â返¤¦Ì回?的Ì?协-议°¨¦包ã¨¹。¡ê

</descr>

<arg>UINT8\_ARRAY</arg> <!-- 需¨¨要°a解a析? -->

</Client::onImportClientMessages>

定义时的518消息号对应的最总处理函数？》

Client\_onRemoteMethodCall 服务器远程调用客户端函数？？

## 事件

事件与消息的区别：

消息用于客户端和服务器之间通信

事件用于客户端处理消息时的回调 （**事件跟网络交互没有直接关系**）

**注册事件（会保存对应的处理函数地址）**

void installEvents()

{

KBEngine.Event.registerOut("onImportClientMessages", this, "onImportClientMessages");

KBEngine.Event.registerOut("onImportServerErrorsDescr", this, "onImportServerErrorsDescr");

KBEngine.Event.registerOut("onImportClientEntityDef", this, "onImportClientEntityDef");

KBEngine.Event.registerOut("onVersionNotMatch", this, "onVersionNotMatch");

KBEngine.Event.registerOut("onScriptVersionNotMatch", this, "onScriptVersionNotMatch");

KBEngine.Event.registerOut("onServerDigest", this, "onServerDigest");

。。。

}

public KBEEventProc()

{

KBEEventProc.inst = this;

KBEngine.Event.registerOut("onEnterWorld", KBEEventProc.inst, "onEnterWorld");

KBEngine.Event.registerOut("onLeaveWorld", KBEEventProc.inst, "onLeaveWorld");

KBEngine.Event.registerOut("set\_HP", KBEEventProc.inst, "set\_HP");

KBEngine.Event.registerOut("set\_MP", KBEEventProc.inst, "set\_MP");

KBEngine.Event.registerOut("set\_HP\_Max", KBEEventProc.inst, "set\_HP\_Max");

。。。

}

**回调：**

Event.fireAll("onVersionNotMatch", new object[]{clientVersion, serverVersion});



Event.fireXXX 消息流动方向 eventsXXX🡪firedEventsXXX🡪doingEventsXXX

最终会调用回调函数

**单独有个线程收发消息+处理事件：**

> Void KBEngine.Event:processInEvents ()+0xa at F:\kbengine\_unity3d\_warring\Assets\Plugins\kbengine\kbengine\_unity3d\_plugins\Event.cs:287 C#

Void KBEngine.KBEngineApp:process ()+0x5 at F:\kbengine\_unity3d\_warring\Assets\Plugins\kbengine\kbengine\_unity3d\_plugins\KBEngine.cs:265 C#

Void KBEngine.KBEThread:run ()+0x19 at F:\kbengine\_unity3d\_warring\Assets\Plugins\kbengine\kbengine\_unity3d\_plugins\KBEngine.cs:34 C#

## 连接状态变化

currserver = "loginapp";

currstate = "autoimport";

currserver = "loginapp";

currstate = "login";

## UI处理

## 初始化

1. 由laoder加载login场景：

void Start () {

**StartCoroutine**(loadInit());

scene.loadScene(true, false);

//loadScene(bool autocreate, bool showProgressbar)