# ResetCore —— Unity3D增强库文档

## AOP

提供最基础的AOP支持。

利用预定义行为与行为队列来进行切片化编程，我们只需要将我们切片的程序写为预定义行为就可以了。

例如：

AQAopManager.Aop

.Log("开始的信息", "结束的信息")

.ShowUseTime()

.Work(() =>

{

//DoSomething

});

我们预定义了Log行为以及计时行为，每当我们需要该类型行为时，只需要建立这个队列就可以进行切片编程。

当然我们也可以通过类似于我们已经创建的AopProxy来创建别的代理类来进行切片编程。

例如：

ActionQueue **testQueue** = AopProxy<ActionQueue>.CreateProxy();

//testQueue.DoSomething()

当我们进行函数调用的时候我们就会调用AopProxy中的PreProceede与PostProceede。

不过这种方法比较简陋，要实现属性形式的额AOP编程需要用到Reflection.Emit。由于涉及到IL语言、动态编译，所以没有进行实现，后期可能会进行进一步开发。

## Asset

ResetCore.Asset实现了Resources与AssetBundle的抽象，并且实现了最简单的AssetBundle框架。我们可以只通过ResourcesLoaderHelper来加载Resource以及AssetBundle下的资源。

我们会先检查AssetBundle目录下是否有相应资源，若没有则会到Resources中寻找。

我们在保存场景的时候，会自动生成一张资源列表，记录了所有资源的名称以及路径。

（这就需要我们保证Resources下面不能出现重名资源，需要遵循一定命名规范）

加载资源的方式如下：

ResourcesLoaderHelper.Instance.LoadAndGetInstance("name.prefab");

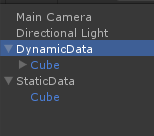
ResourcesLoaderHelper.Instance.LoadResource("xx.xx")

ResourcesLoaderHelper.Instance.LoadResource<MyType>("xx.xx")

我们还实现了场景的xml序列化以及反序列化。



场景中需要有DynamicData与StaticData两个文件夹来装载动态加载场景物体



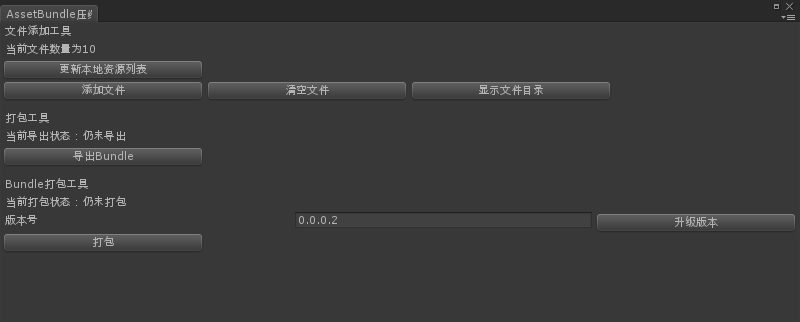
导出Xml至PathConfig下定义的Xml中。

当我们需要加载场景的时候，我们只需要调用

XmlSceneBuilder.Instance.SceneBuilder("sceneName", (*comp*)=> { })

来进行场景的加载。

我们还支持AssetBundle的打包行为



打开“Windows -> AssetBundle打包器”我们就可以看到这样的界面。

我们可以将新文件添加到打包列表当中，每次打包时版本号都会自动升级。

最终导出文件会到工程目录下的AssetBundleExport文件夹下。

下载完成并且解压后只需要放入AssetBundle文件夹下即可进行对AssetBundle的同步读取。

EncryptHelper还对AssetBundle加密进行了支持，不过尚未纳入整体框架之中。

StreamingDataLoader支持流媒体文件夹下文件的读取。

DownloadManager支持断点下载，Version支持版本号的最基本功能。

原本计划内打算实现挂载器的框架，不过后面由于种种原因没有完成，后续版本中或许会出现吧。

AssetBundle最基本框架有了，但是暂未将下载与网络验证纳入框架，因为不同的项目要求都不一样，所以需要在具体项目中进行构架。

## BehaviorTree

行为树，尚属于测试阶段。（一直没有测试。

将BehaviorRoot挂载到GameObject上，并且输入相应的Xml地址，就可以实现AI。

## CodeGener

基于CodeDom进行的封装，降低了使用CodeDom带来的开发复杂度。

例如：

CodeGener **testGener** = new CodeGener("namespaceName", "className");

**testGener**.AddBaseType("BaseType");

**testGener**.AddImport("System", "UnityEngine");

**testGener**.AddMemberField(typeof(int), "varName", (member) =>

{

member.AddComment("我是注释");

member.AddFieldMemberInit("1");//初始化表达式

});

//....

**testGener**.GenCSharp("Path")

极大简化了动态生成代码的过程。在许多数据源转类型代码的功能块（特别是数据抽象化那块），得到了广泛应用。

## CSTool

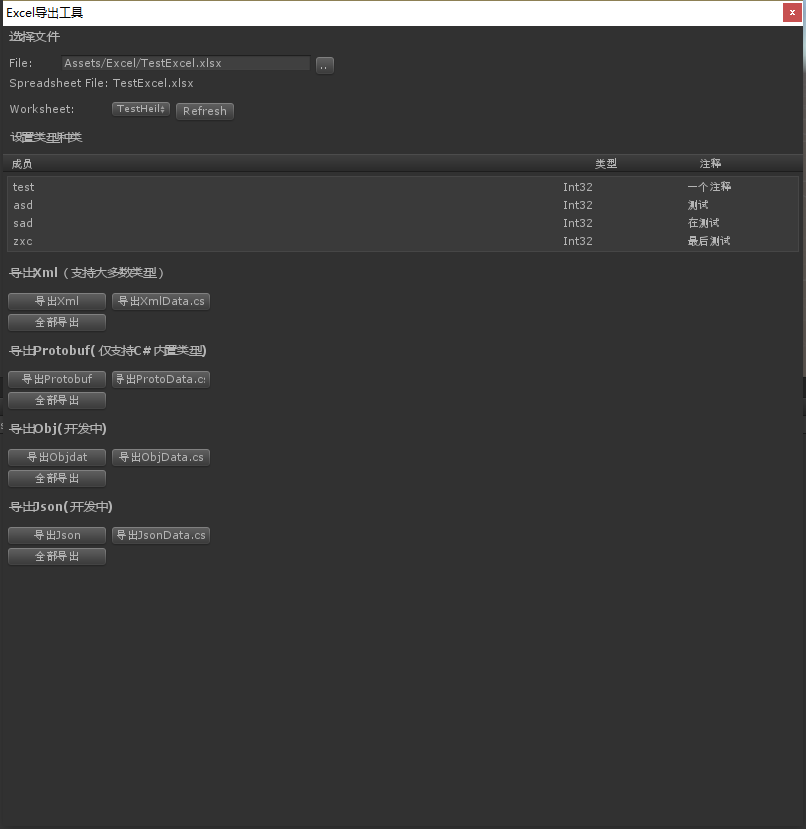
通过运行外部的程序来实现一些功能，之前拿该工具实现读取Excel的功能，现在改用NPOI已经不用该工具了。不过后期有别的功能需要实现的时候或许会再捡起来使用。

CSToolLuncher.LaunchCsToolExe("command..")

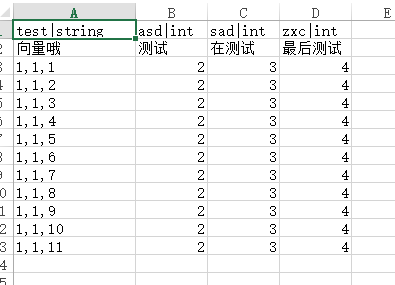
在同目录的说明中会提及有哪些指令

## DataGener

将数据进行抽象化，同时生成一个数据文件以及一个用于访问的类文件。



Excel当然也有格式要求



第一行为“变量名|类型名”

第二行为注释

后面的都为数据。

一般数据访问方式为下面形式：

TestHeihei.dataMap[1].**test**

（索引从1开始）

用户不需要关心到底数据源为Xml或者是Protobuf，只管生成之后调用就行了。

目前该读取器只支持Xml与Protobuf两种序列化方式，后期会开发ScriptableObject以及Json两种新格式。

数据以及CS文件储存路径都在PathConfig中进行了定义。

## DataStruct

数据结构的一些类型。

主要用于学习算法。实用性不强。

## Debug

用于提供调试功能。

目前只有现实FPS的工具类。

## DllManager

用于管理Dll导入以及C++交互。

还没怎么进行开发。

## Editor

关于Editor中一些通用的东西。

## Events

事件分发系统。

支持单播以及广播。

EventDispatcher.AddEventListener("xxx", act);

EventDispatcher.RemoveEventListener("xxx", act);

EventDispatcher.TriggerEvent("xxx")

当绑定对象时，则可支持单播。

EventDispatcher.AddEventListener("xxx", act, object);

EventDispatcher.TriggerEvent("xxx", object)

上面的代码就是单播的例子，消息只会发送给object。

最近版本中追加了数据绑定模块。

Text **txt**;

**txt**.*text*.BindData<string>("xxx", (*str*) => { **txt**.*text* = *str*; });

DataBind.ChangeData<string>("xxx", "newValue")

上面展示了数据绑定以及改变数据的过程。

## FSM

有限状态机。

该部分非原创，感觉Github上面写得挺好就为己所用了。

只需要创建一个枚举类型，每个类型代表了一个状态，然后在Monobehavior中编写回调函数即可。

## GameSystem

部分游戏系统：目前包含了Buff系统

（其实这种非通用系统，写进增强库不是很合理。

## ImportHelper

提供了导入的回调函数，可以根据要求进行填充。

原本想做成一个面板，不过感觉还是需要根据项目需要去填写，所以现在也没有做。

## Libs

一些库。

## Lua

对Lua进行了统一的管理，使用了uLua作为基础。支持进行了池的管理。

编写了LuaComponent，能够在GameObject上用LuaComponent控制整个MonoBehavior流程。LuaManager则是控制了Lua的调用。

LuaManager.instance.Call("fileName", "functionName", arg1, arg2, ..)//调用文件中的函数

LuaManager.instance.DoLua("fileName")//运行Lua

LuaManager.instance.LoadLua("fileName")//加载文件中的模块

ModManager则是在沙盒目录下控制了Lua的运行。

## MySQL

由于很少直接用客户端控制SQL所以只做了简单的数据库连接，没有对操作进行封装。

MySQLManager.OpenSql("host", "database", "id", "password", "port")

MySQLManager.ExecuteQuery("command..")

仅仅对打开数据库与执行语句进行了支持。

## NetPost

对Http进行了简单的封装。

我们只需要继承BaseNetTask，就可以通过发送自定义的Task来完成网络任务。

NetTaskDispatcher.instance.AddNetPostTask(new ExampleNetTask(), "queueName")

基类我们需要实现初始化以及回调就可以了。

namespace ResetCore.NetPost

{

public class ExampleNetTask : NetPostTask

{

public ExampleNetTask(*Dictionary*<string, object> **taskParams**, Action<JsonData> **finishCall** = null, Action<float> **progressCall** = null)

: base(**taskParams**, **finishCall**, **progressCall**)

{

}

public override string taskId

{

get { return TaskId.TEST\_TASK; }

}

protected override void OnStart()

{

base.OnStart();

}

protected override void OnProgress(float **progress**)

{

base.OnProgress(**progress**);

}

protected override void OnFinish(LitJson.JsonData **backJsonData**)

{

base.OnFinish(**backJsonData**);

Debug.*logger*.Log(**backJsonData**.ToString());

}

}

}

## NGUI

支持最基础的GUI层级管理以及动态加载与卸载。

UIManager.Instance.ShowUI(UIConst.UIName.name, *arg*, afterAct)

UIManager.Instance.HideUI()

要实现UI的子类也只需要进行对函数的重载即可。

GUI被分成三层，NormalUI，PopUpUI，TopUI。

## Object

一些场景常用物体的控制，如音频，摄像机，粒子。

目前最完善的只有音频管理器，利用池来控制音效的播放并且支持导入到Mixer中。

AudioManager.Instance.PlayObjectSE(*GameObject*, ClipName, MixGroup)//绑定物体音效

AudioManager.Instance.PlayGlobalSE(*GameObject*, ClipName, MixGroup)//全局音效

AudioManager.Instance.PlayBGM(ClipName)//播放BGM

## PlatformHelper

针对不同平台导出的工具，如一键打包，或者多语言交互。

尚未开发阶段。

## Plugins

一些外部库。

## Shader

Shader，尚未开发。

## Test

测试用。

## UGUI

与NGUI对应，详见NGUI部分。

## Util

常用的工具类。

Const：

中有常用的常量。

协程管理器：

CoroutineTaskManager.Instance.AddTask(*IEnumerator*, CallBack, BindObject, AutoStart)

里面提供了各种方便的工具，例如等待若干秒之后执行、循环执行。

最主要是支持协程的暂停与恢复。以及通过名字进行获取。

Extension：

各类类型的扩展，其中String类型的扩展较为重要。

ObjectPool：

物体池与特效池。

ActionQueue：

行为队列。

还有单例模板、场景加载辅助类、线程辅助类等等。

## Xml

Xml读取增强库。

**xDoc**.ReadValueFromXML<int>(nodeNames, defValue)

**xDoc**.WriteValue(nodeNames, value)

xDoc.Save(Path)

对XDocument与XElement进行扩展简化了读取与写入操作。

尽量少用，而使用抽象数据多一些。