框架介绍：

**一、日志系统：**DebugOutput.cs 接收Log日志并输出 MyDebug.cs 保存指定Log日志到本地

**二、ToLua：**

tolua#是Unity静态绑定lua的一个解决方案，它通过C#提供的反射信息分析代码并生成包装的类。它是一个用来简化在C#中集成lua的插件，可以自动生成用于在lua中访问Unity的绑定代码，并把C#中的常量、变量、函数、属性、类以及枚举暴露给lua。它是从cstolua衍变而来。从它的名字可以看出，它是集成了原来的tolua代码通过二次封装写了一个C#与tolua(c)的一个中间层。

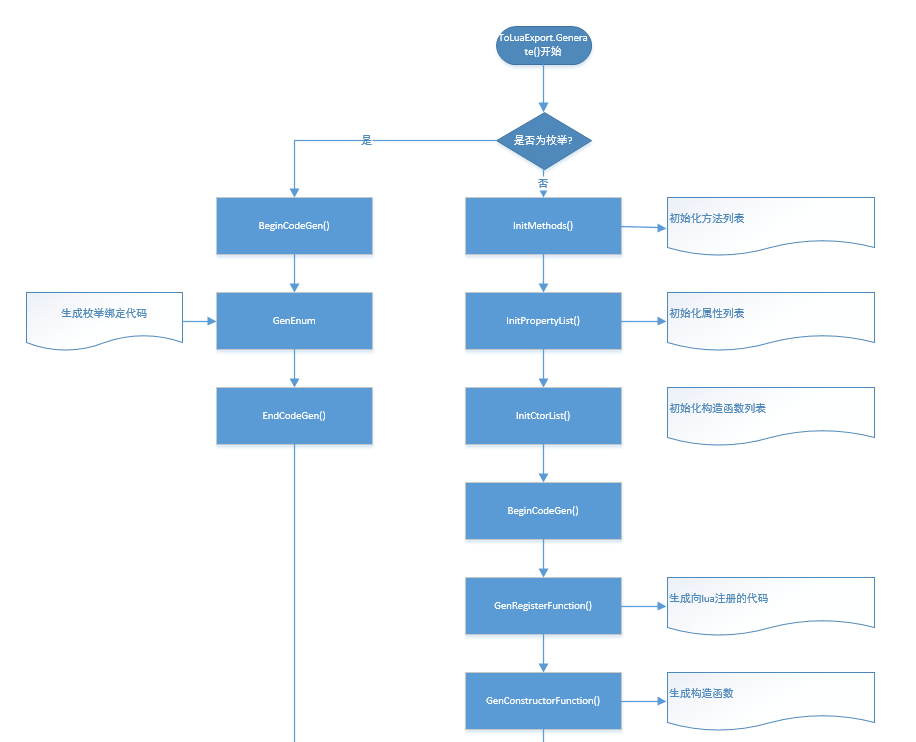
**1.Generate All（ClearLuaWraps（））流程:**

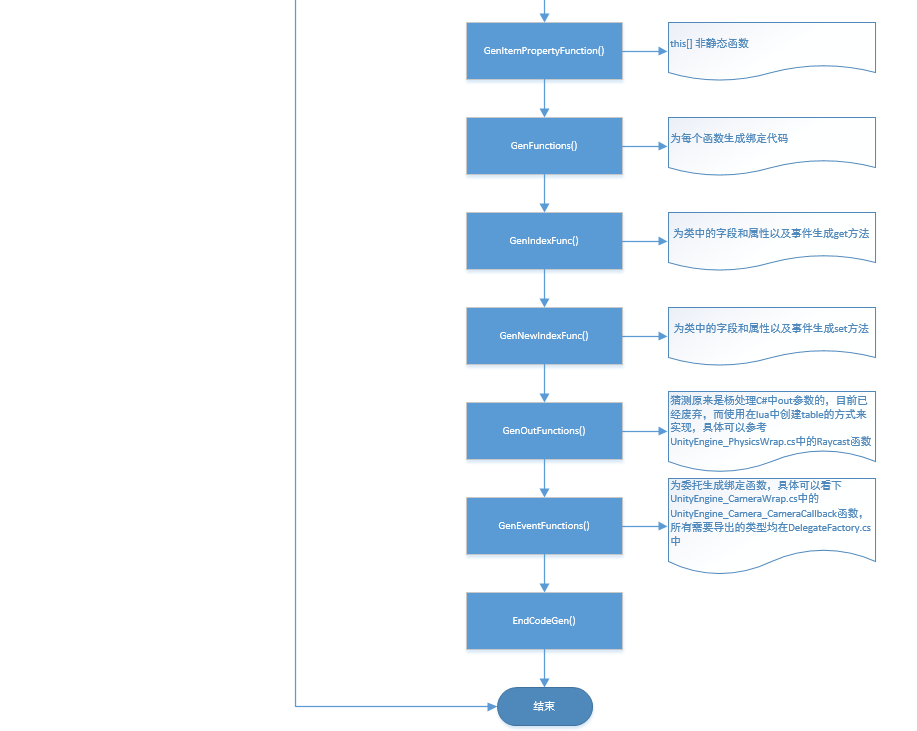
(1).删除Source/Generate下的所有Wrap文件.

(2).调用ToLuaExport.GenDelegates()创建DelegateFactory.cs文件；紧接着创建LuaBinder.cs文件。

(3).调用GenLuaDelegates（），生成委托绑定的代码，它会从CustomSettings. customDelegateList里面取出所有自定义导出的委托列表，然后把CustomSettings.customTypeList里面的所有类型根据一定规则加入到list中，最后调用ToLuaExport.GenDelegates()方法来生成委托绑定的代码，生成的代码在DelegateFactory.cs中。

(4).调用GenerateClassWraps（），主要是调用ToLuaExport.Generate()方法来生成类的绑定代码。其中ToLuaExport.Generate()，若为Enum，那么直接调用GenEnum()；其他类则GenFunctions() => 为类中的每个函数的生成绑定代码；GenIndexFunc() => 为类中的字段和属性以及事件生成get方法；GenNewIndexFunc() => 为类中的字段和属性以及事件生成set方法。





(5).调用GenLuaBinder（），生成向lua注册C#类的绑定代码，这个代码存放在LuaBinder.cs文件中，这部分代码中不包含BaseType（Array Enum Object String 等等）的注册。Bind函数是向lua虚拟机注册所有的wrap类。这里BeginModule可以理解为命名空间，在lua中以table形式实现。BeginModule（null）代表当前命名空间是global全局域，而BeginModule（“UnityEngine“）则代表后面都注册到\_G[‘’UnityEngine”]表中。那么在基础的wrap文件中，BeginClass类似BeginModule，但实际功能比BeginMould更加复杂，并且传递了第二个可选的参数，代表着class的基类，在c层以metatatable的形式实现对C#继承的支持。RegFunction是向该class注册一个方法，RegVar是向该class注册一个属性。

**2.ToLua使用：**

* 提供Lua-c#值类型、对象类型转化操作交互层。（ObjectTranslator.cs、LuaFunction.cs、LuaTable.cs、ToLua.cs等）
* 提供Lua虚拟机创建、启动、销毁，Require、DoFile、DoString、Traceback等相关支持。（LuaState.cs、LuaStatic.cs）
* 提供导出工具，利用c#反射，对指定的c#类生成对应的wrap文件，启动后将所有wrap文件注册到lua虚拟机中。（ToLuaMenu.cs、ToLuaExport.cs、ToLuaTree.cs、LuaBinder.cs、CustomSetting.cs等）
* 提供c#对象和lua userdata对应关系，使该userdata能访问对应c#对象属性，调用对应c#对象函数。lua支持一定的面向对象(类、继承)。管理这些对象的内存分配与生命周期、GC。(LuaState.cs）
* 提供支持功能Lua Coroutine、反射等，Lua层重写部分性能有问题对象如Vector系列。（Vector3.lua等）

**在实际项目中，还需要把项目中我们写的lua文件进行加载，可以使用LuaState.DoFile、LuaState.DoString，也可以在lua文件中进行require。但无论何种方式，都要格外注意搜索路径，如果使用LuaState中的DoFile和DoString方法，要调用相应的LuaState.AddSearchPath(“path”)加入进去。**

**更多参考资料：**

<https://www.jianshu.com/p/ef1e2641e0c6>

<https://blog.csdn.net/xingaaaxing/article/details/72869709>

<https://blog.csdn.net/suifcd/article/details/78632177>

<https://www.jianshu.com/p/3a45928ca14a>

**三、资源打包：**（**注：除Lua文件放置LuaScripts中，其他要打包的资源需放置data目录下**）

**打包操作流程：**可一键打包（MyAssetsBundle 🡺BuildAll（打包相关属性都是默认设置））。也可自定义打包，先选择Assets/CheckBuildPattern/ShowOrUpdateBuildPattern**生成打包模式配置文件（buildPattern）**，然后查看其中的选项是否符合预期，然后Assets/BuildManifest生成对应资源配置文件，可查看当前打包信息，最后BuildAll。

**打包总体流程：**调整或生成打包模式（BuildPackPattern） 🡺 根据不同的打包模式，build相应的AssetBundle（lua是另外一套除外） 🡺 根据不同打包模式生成相应的配置文件 🡺 根据相应平台进行BuildAssetBundle。

1. MyAssetBundleMenu.cs。打包入口
2. BaseBuild.cs。打包基类
3. 设置各类打包属性：

**searchPath**：文件搜索路径

**searchPattern**：文件匹配类型

**option**：指定搜索操作是应包含所有子目录（AllDirectories）还是仅包含当前

目录（TopDirectoryOnly）

**bundleName**：AssetBundle后缀（暂时不用）

**assetFolderName**：资源目录名称

**BuildType**：根据打包资源不同暂定为两类打包方式：第一是资源打包，目前根据粒度可拆分为3中方式，BuildAssetsWithAssetBundleName（粒度最大，包体最小）；

BuildAssetsWithDirectroyName（粒度适中，包体适中）；BuildAssetsWithFilename（粒度最小，包体最大），初步构想，没有依赖且体积适中或偏小的资源采用BuildAssetsWithAssetBundleName方式打包，例如特效等；存在依赖并资源较多的采用BuildAssetsWithDirectroyName，例如界面的Atlas资源；依赖关系多并体积较大，采用BuildAssetsWithFilename，例如Prefab。第二是Lua打包，因为要生成bytes文件，所以是单独与资源打包，但总体采用的是BuildAssetsWithDirectroyName去打包。

1. 记录打包信息：

**builds**:记录当前需要打包的所有assetBundle资源

**packedAssets**:记录当前已经添加的assetBundle资源。防止打包重复资源

**patterns**：记录当前选择的所有打包模式

**allDependencies**：记录当前添加的assetBunlde资源的依赖信息

1. GetBuilds()为整个打包过程的总入口。添加配置文件（Manifest）信息，这里配置文件存在与否不关键，只是先添加assetBundle信息。
2. 添加打包方式（buildPattern）信息，若不存在那么会生成默认的打包方式（暂定图片资源打包采用BuildAssetsWithDirectroyName，预制体打包采用BuildAssetsWithFilename）。
3. 根据反射获取对应的打包脚本，并设置对应打包属性。
4. 缓存当前所有的asset对应的依赖文件。
5. 根据特定的打包方式开始打包资源。（打包过程：不管那种打包方式，都必须先执行这两步，首先获取当前searchPath下的所有资源，但会剔除已经打包过的资源，然后，获取对应资源的依赖资源，根据allDependencies[assetPath].Count == 1，剔除已被打包的依赖资源。）
6. 至此获取了所有需要打包的资源，然后根据当前所有的资源信息生成配置文件，并打包该配置文件，开始根据平台执行打包。打包输出目录是根据当前平台生成路径AssetBundles/XXXX。

**AssetBundle理解：**

1. (常变化的采用细粒度打包，不常变化，但依赖会改变的采用适中粒度打包，资源较小且体积小采用粗粒度打包)AB粒度不宜过细（一个资源一个AB），因为一方面导致加载IO次数过多，从而增大了硬件的耗能和发热的能力，其次每个AB的加载，在Android平台不Unload的情况下，其每个文件序列化所占的内存为512KB，当大量贮存AB文件时，内存占用会显得非常巨大。
2. 资源目录规划要做动静分离处理，不变的遵从把游戏中同时使用的资源放到同一个目录，不同类型的可以区别对待。，会变的分到另外一个目录，内部也按前面的方式对资源进行目录安置。动静分离，公共资源（动）与界面资源（静）分开。就是说随项目扩张的资源为动，例如界面等资源属于静 。 sprite 和 texture 随项目进度增加的资源（例如头像Icon，物品Icon,卡牌立绘），这类资源不建议使用Sprite并打包图集，这样会导致图集所占内存越来越大（加载一张图片时，会加载整个图集），虽然能降低DrawCall（其实也降低不了多少，因为当一张图集打过2048，unity会自动合到另外一张图集中去，drawcall还是上去了），但得不偿失。所以建议使用Texture，单独加载单独的drawcall，折中的方法。（取舍问题，合图集，一定程度上减少drawcall，不合图集，内存会减少一些）。
3. 将可能同时加载的对象分组。例如模型，纹理和动画等。
4. 若多个文件的依赖项都相同（材质，sharder等），那么可考虑将将依赖项单独打包，并提前解压。
5. 如果该包中有一半资源很少同时加载，考虑拆分包。
6. 考虑合并内容经常更新的小型包。

**总结：**

1. 打包总结：首先资源分配方面：先根据类型分配好资源目录，做好资源的动静分离，多依赖的公共资源单独存放，无依赖且体积小的资源单独存放，无依赖但整体体积较大资源根据需求来取舍。然后打包方面：剔除已添加build的资源（避免资源重复打包），维护资源依赖关系（但要排除多依赖打包问题），选择合适的打包方式进行打包。
2. 优化包体：1.检查资源重复 2.根据类型采取不同粒度打包方式 3.检查资源（例如图片的Generate Mip Map）4.检查资源包编译选项（BuildAssetBundleOptions）。

**四、资源加载：**（**直接使用项目资源或启用AssetBundle加载模式**）

自动处理资源的依赖，同时添加资源的引用计数，让同一份资源不会重复加载亦不会轻易卸载。支持开发模式和AssetBundle模式（开关useEditorPrefab即可实现）。加载文件只需传入资源名称即可。

框架介绍：

1. **入口：**ResourceManager.Init（），先初始化配置文件以及初始化AssetBundle和对应的依赖文件（AssetBundleManifest）。
2. **加载资源部分：**

MyAsset：单个资源对象。

**Callback**：资源加载后的回调

**assetName**：资源名称

**assetType**：资源类型

**asset**：以及加载完成的资源

**isDone**：资源是否加载完成

**references**：当前资源的引用数量

MyAssets:资源管理类

提供资源的加载与卸载，维护资源的引用

MyBundle：单个Bundle对象。

**Path**:记载路径

**Name**：名称

**assetBundle**：Bundle资源

**error**：加载是否有误

**isDone**：是否加载完成

**references**：Bundle引用数量

**dependencies**：依赖文件

MyBundles:Bundle资源管理类

Bunlde的加载及卸载，管理Bundle的依赖文件的下载及卸载，维护引用数量。

主要提供直接记载与异步加载两种方式，直接加载只是维护引用关系，并会返回对象。异步需使用回调函数来处理后续。（异步方式没有使用协程）

1. **卸载资源部分：**

MyAssets中assets会缓存所有当前加载的资源，在Update中会根据引用数量来执行卸载操作，若当前引用数量小于0，那么执行UnLoad()，释放指定的Asset，进一步调用OnUnload()更新Bundle的引用数量，若Bunlde引用也为0，那么也删除该Bundle。

**注意：加载过程首先初始化目录AssetBundle及其对应的AssetBundleManifest文件，然后加载资源配置文件。**

**资源加载与释放理解：**

**1.加载：**

**AssetBundle运行时加载，**LoadFromFile等于是把硬盘或者网络的一个文件读到内不能一个区域，这时候只是个AssetBundle内存镜像数据快，还没有Asset概念。

**Asset运行时加载，**LoadAsset这才会从AssetBundle的内存镜像里读取并创建一个Asset对象，创建Asset对象同时也会分配相应内存用于存放（反序列化的过程）。

**Instantiate（object)，**克隆（clone）一个object的完整结构，包括其所有Component和子物体。

**2.释放：**

**AssetBundle.Unload(false),**是释放AssetBundle文件的内存镜像，不包含Load创建的内存对象；**bundle内的序列化数据将被释放，但是任何从这个bundle中实例化的物体都将完好，所以不能从这个bundle中加载更多物体（应设置为Null）。注：确定这个AssetBundle是否需要反复读取，若不是，那么可以在AssetBundle.Load之后立即执行Unload（false）。**

**AssetBundle.Unload(true)，**是释放AssetBundle文件内存镜像和并销毁所有用Load创建的Asset内存对象。**所有从该bundle中加载的物体也将被销毁，如果场景中有物体引用该资源，引用会丢失（应设置为Null）。**

**Destroy,**主要用于销毁克隆对象，也可以用于场景内的静态物体，不会自动释放该对象的所有引用。虽然也可以用于Asset，但概念不一样，如果用于销毁从文件加载的Asset对象会销毁相应的资源文件。但如果要销毁的Asset是Copy的或者用脚本动态生成的，只会销毁内存对象。等到没有任何 游戏场景物体在用这些Assets以后，这些assets就成了没有引用的游离数据块了，是UnusedAssets了，这时候就可以通过 Resources.UnloadUnusedAssets来释放,Destroy不能完成这个任 务，AssetBundle.Unload(false)也不行，AssetBundle.Unload(true)可以但不安全，除非你很清楚没有任何 对象在用这些Assets了。

**Reources.UnloadAsset(object),**显式的释放已加载的Asset对象，只能卸载磁盘文件加载的Asset对象。

**Resource.UnloadUnusedAssets，**用于释放所有没有引用的Asset对象。

**GC.Collect（），**强制垃圾收集器，立即释放内存。

**几种加载方式的区别：**

一是静态引用，建一个public的变量，在Inspector里把prefab拉上去，用的时候Instantitate。

二是Resource.Load，Load以后instantiate。

三是AssetBundle.Load，Load以后instantiate。

三种方式有细节差异，前两种加载，引用对象例如Texture是在instantiate加载，而AssetBundle.Load会把prefab的全部asset都加载，instantiate时只是生产Clone。所以前两种方式，除非你提前加载相关引用对象，否则第一次instantiate时会包含加载引用asset的操作，导致第一次加载的滞慢。

**资源加载的优化相关：**

1. **第一次实例化一个Prefab。**这时候都会卡一下，因为在第一次Instantiate之前，相应的Asset对象还没有被创建，要加载系统内置的AssetBundle并创建Assets，第一次以后虽然Destroy了，但prefab的Assets对象还在内存里，那么第二次自然就快了。
2. **加载速度与内存大小的取舍问题。**若是为了让界面打开速度达到最快，那么在加载AssetBundle并加载完成Asset之后，不选择Resource.UnloadAsset和AssetBundle.Unload，那么该资源和AssetBundle都会一直存在内存中，那么下次打开就不会再次加载。如果为了优化内存，一种是方式时，加载完AssetBundle和Asset之后，执行AssetBundle.Unload（false），这样可以省一部分内存（但注意Unload之后，就不能在加载其中的内容了）；另一种方式时，资源使用完毕之后，执行AssetBundle.Unload（true）和Resource.UnloadUnusedAssets，从内存中清除AssetBundle和对应Asset。注：其实也可以清除Asset保留AssetBundle，所以会有取舍问题。
3. **资源卸载误区。**例如现在从 某个AssetBundle例Load了一个prefab并克隆出来，

obj = instantiate（AssetBundle.load（“prefab”）），这个prefab比如是个NPC，在不需要的时候使用了，Destroy（obj），以为卸载干净了，其实这个时候只是释放了Clone对象，那些通过Load加载的所有引用、非引用Assets都还存在于内存当中，这种时候应该在Destroy以后，再用AssetBundle.Unload（true）来彻底的释放干净。

1. **多资源引用**

例如从磁盘中读取了一个文件“test”到内存并建立了一个AssetBundle对象ab；然后再从ab例读取并创建了一个texture tx1，游戏中有个游戏物体obj1的某个贴图指向它，然后另一个游戏物体obj2的某个贴图也指向了它。这个时候若使用**Unload（true）**，那么obj1和obj2都会丢失贴图引用，变成黑色的，因为指向的texture tx1已经不能存在了；那么使用**Unload（false）**，那obj1和obj2不会改变，只是该ab的内存镜像释放了；再使用Destroy（obj1），obj1被释放了，但是并不会释放Load的asset资源（即tx1），那么继续使用Resource.UnloadUnusedAssets时，其实并不会释放内存资源，因为tx1还在被obj2使用，再继续Destroy（obj2），obj2释放，但还是不会释放tx1，这时候继续调用Resource.UnloadUnusedAssets，因为tx1不存在引用了，所以就会释放在内存中的tx1。**注：UnusedAssets不但没有被实际物体引用，也要没有被生命周期的变量引用，才可以理解为Unused（引用计数为0）。且如果你用个全局变量保存你Load的Assets，又没有显式的设为Null，那在这个变量失效前无论如何UnloadUnusedAssets也释放不了这些Assets的。如果你这些Assets又不是从磁盘加载的，那除了UnloadUnusedAssets或者加载场景以外没有其他方式可以卸载。若存在static的Assets资源，那么就必须手动的释放该资源，不然会一直存在内存中。**

**五、资源更新(HotManager)：**

**比对本地文件（file.txt）与远端文件（file.txt）的差异，首先检测版本文件是否有差异，有差异才可开始更新，无差异直接进入游戏（即更新完毕）。若可更新，那么只需比对剩余文件的异同（检测MD5），选择更新即可。**

**思路：**

1. 首先检测本地是否有file.txt配置文件，若不存在，那么获取包体中的配置文件，若包体也不存在，那么从远端下载。若存在本地文件，读取并缓存文件信息。
2. 开始获取远端文件，首先下载并解析远端file.txt配置文件，开始于本地文件比对，若两个文件的MD5码存在差异，那么可以下载远端文件来替换本地文件。
3. 开始下载需求的远端文件。下载过程：先将需要下载的文件添加至下载队列（信息包括：远端文件路径及本地路径），然后在ThreadManager中的OnUpdate方法中，检测到下载队列有文件，那么开始调用下载方法OnDownloadFile，使用WebClient从远端路径下载至本地路径，下载完成后执行回调，保存已下载的文件（防止重复下载）。
4. 最后更新配置文件，完成更新。

**注意：**

1. **在安卓平台下，首次打开游戏是获取不到本地文件的，所以需要新建目录，并从远端下载所有文件防止该目录，第二次就可从本地读取。**
2. **注意各平台的包体资源路径（Android：**Application.dataPath + "!/assets/"，**IOS：**Application.dataPath + "/Raw/"，**PC：**Application.dataPath + "/StreamingAssets/"**）**
3. **各平台的读写目录：**

**Application .streamingAssetsPath（只读目录）**

**Application.persistentDataPath（读写目录）**

**IOS:**

Application.dataPath：Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/xxx.app/Data

Application.streamingAssetsPath :  Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/xxx.app/Data/Raw

Application.persistentDataPath :   Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/Documents

Application.temporaryCachePath :   Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/Library/Caches

**Android:**

Application.dataPath : /data/app/xxx.xxx.xxx.apk

Application.streamingAssetsPath :jar:file:///data/app/xxx.xxx.xxx.apk/!/assets

Application.persistentDataPath : /data/data/xxx.xxx.xxx/files

Application.temporaryCachePath : /data/data/xxx.xxx.xxx/cache

**Windows:**

Application.dataPath : /Assets

Application.streamingAssetsPath : /Assets/StreamingAssets

Application.persistentDataPath :C:/Users/xxxx/AppData/LocalLow/CompanyName/ProductName

Application.temporaryCachePath :C:/Users/xxxx/AppData/Local/Temp/CompanyName/ProductName

**Mac:**

Application.dataPath :    /Assets

Application.streamingAssetsPath : /Assets/StreamingAssets

Application.persistentDataPath :/Users/xxxx/Library/Caches/CompanyName/Product Name

Application.temporaryCachePath :/var/folders/57/6b4\_9w8113x2fsmzx\_yhrhvh0000gn/T/CompanyName/Product Name

**六、状态机：**

**操作流程：添加状态StateMachine.Add(XXXState)** => **构造函数初始化XXXState类**（例：public CheckStage(FrameworkMain owner) : base((int) MyStage.check, owner{})，主要是方便访问共有类及确定当前状态ID） => **切换状态只需调用StateMachine.ChangeState（ID）**（指定状态的id）

**1.总基类**：StateBase。

Owner：使派生类都持有某个类的引用

Id：当前状态ID

IsFinished：检测当前状态是否已经完成（若已完成是不能执行OnAfterEnter（））

BeforeEnter（）：进入状态前

OnEnter（）：进入状态

OnAfterEnter（）：进入状态后

OnRunning（）：状态运行中

OnExit（）:状态退出

IsValidChange（）：检测状态是否有效

**2.派生基类（扩展类）：**BaseState（不一定是这个），可以自定义不同的派生基类，为了方便扩展不同的类型状态，不用类型可持用不同类的引用。

**3.状态机：**StateMachine。状态管理

\_states：记录所有状态

\_currState：获取当前正在进行的状态

\_lastState：刚刚结束的状态（上一状态）

Add（）：添加状态

Remove（）：删除指定状态。通过状态ID

RemoveAll（）：清空所有状态

ExitCurrState（）：退出当前状态

ChangeState（）：切换状态。退出当前状态，执行指定状态

OnRunning（）：持续运行当前状态

Find（）：查询某个状态，通过状态ID

**七、事件传递(EventDispatchCenter)：**

**1.事件注册：**LTEventCenter. Regist（string notifyID, Action<object> action, bool inQueue），**notifyID**：确定事件ID，**action**：事件回调，**inQueue**：是否按照队列执行（若为true，那么就是按队列依次执行，处理一个事件多次注册并且保持注册的先后顺序来执行回调；若为false，则立即执行，不在乎先后顺序）。根据是否依次执行，事件回调存于不同集合（true：\_queueEventMap；false：\_eventMap）

**2.事件派发：**LTEventCenter. SendNotify（string notifyID, object obj），**notifyID**：确定事件ID，**obj**：回调参数。派发的时候，检测\_queueEventMap或\_eventMap那个有值就执行那个集合中的所有回调。

**3.取消事件注册：**LTEventCenter. UnRegist（string notifyID, Action<object> action），删除指定事件ID的回调。