协程Coroutine

Coroutine

- 基本概念和设计目的
  - 游戏开发中会遇到
    - 并行逻辑:如同时加载多个资源
    - 需要多帧执行的延续性逻辑:如寻路到某一点
    - 同步这些并行和延续性逻辑:如现寻路到某一点,说一句话,在寻路到某一点
  - 目的
    - 轻型的并行概念
    - 可以同步并行逻辑
    - 避免多线程带来的逻辑麻烦
  - 概念
    - 假的多线程概念
    - 可以在某些点同步
    - 可以对将复杂逻辑分帧,可以在主线程内非阻塞的运行一个持续性逻辑
  - 一个例子
    - 写lemurator
    - startcoroutine
    - 要由mono发起
- 实现原理
  - c# 迭代器,将一段逻辑分解到迭代器中
  - 另一个udate(),在每个update中执行一次迭代器,看一下mono的生命周期
  - 每次yield 都会离开本次执行 ,迭代器会记住这个位置和上下文 ,下一次update到这个携程从上次离开的位置机型执行
- coroutine和多线程
  - coroutine 不是多线程
  - coroutine 是在主线程内每帧update,假的并行
  - 因为不是多线程,所以不用考虑变量的访问时序
- 常用的几种coroutine操作
  - yiled return null:等到下一帧继续执行
  - yield return new waitforendofframe: 这帧结束后继续执行
  - yield return new waitforfixedupdates:N 帧后继续执行
  - yield return new waitforseconds:N 秒后继续执行
  - yield return www:开启WWW的异步下载,直道下载完毕后继续执行(这里面的下载过程实际上会调用真的多线程)
  - yield coroutine: 等待另一个协成 ,这时把协程串联起来的关键
- 执行效率和时机
  - 一个什么也不做的携程也至少需要在等待一帧来完成(即最好执行2帧)
  - yield return null 和yield return new waitforendofframe: 的时机不同
  - 避免多个 coroutine 连接时无谓的多帧执行
    - coroutine中的分支(某些情况可以不等)
    - 某些条件下可以不用协程就另外写一套非协程的代码
- 实际应用:定时器
  - 固定时间后运行
  - 定期运行
  - 当然还有非携程做法 :如使用 update 和invoke
- 实际应用:将复杂操作分帧
  - 生成一张贴图的灰度版本
- 实际应用:控制互斥区
  - 保证一段代码只同时被一个协程进入
- 实际应用:串联几个持续性逻辑
  - 控制怪物先导航到某一处,然后做某个动作,再导航到某一处
- 常见使用误区
  - 协程都在一个主线程里
  - 必须依赖于一个启动它的mono,这个mono disable或者destroy这个协程都会终止
  - 为了逻辑清晰 ,尽量不用 stopcoroutine
  - 可以用 iemurator 的movenext 查询携程是否还在运行
  - 不要滥用携程 这不是容易阅读的代码

资源加载

- Unity的资源类型和对资源的管理方式
  - 类型
    - Asset
      - (pure asset)mat tex mesh animator shader ...
      - gameobject(预置),必须instantiate才能使用
      - scene
      - scriptableobject(custom)
      - textasset
      - built-in
  - 编辑器内管理
    - assetdatabase
  - 运行时管理
    - resource
    - assetbundle
- AssetDatabase 接口
  - 编辑器内加载,效率高 速度快
  - 加载
  - 卸载
  - 创建一个 asset
  - 修改并保存一个已有 asset
- Resources 接口
  - 运行时加载
  - unity自动控制资源的管理
  - 效率高
  - 资源需要放在 Resource 文件夹下
  - 加载
  - 卸载
  - Unity 会保证资源的引用计数
  - 防止你的资源无法被 gc!
  - 无法精确控制你的资源的卸载,难以在线更新某个资源
- AssetBundle
  - 简介
    - Assetbundle 的设计原理
      - 资源分包
        - 一个资源可以分到多个包里
        - 多个资源可以打进一个包
      - 可以实时从任意位置加载资源包
      - asset 在打包时维护一个指向它所依赖的包的 id
    - 运行时加载 将资源打成包 自己控制整个资源的管理
    - 加载效率稍差
    - 要自己控制整个打包和资源的管理,包括资源的引用计数
    - 对资源的管理更加精确 ,可以在线更新某个资源
  - 制作AssetBundle
    - BuildPipeline.BuildAssetBundles
  - 从AssetBundle 加载资源
    - www 接口
    - assetbundle 接口
  - 同步加载和异步加载
  - 使用AssetBundle 管理资源之间的依赖
    - 如何处理一个资源被共用的问题
    - 打包堆栈
    - 堆栈上的已有资源不会被重新打包
    - 被依赖的bundle需要先加载
  - 使用AssetBundle 加载场景
    - BuildPipeline.BuildPlayer
    - LoadLevel/LoadLevelAsync
  - 资源的生命周期和内存管理
    - 生命周期
      - AssetBundle disk img(WWW/WWW.dispose)
      - AssetBundle Released Raw(WWW.assetbundle/AssetBundle.Unload())
      - Asset(保存了对bundle的引用id)
    - 内存管理
      - AssetBundle.unload(true)
      - AssetBundle.unload(false)+Resources.UnloadUnusedAssetBundle+DestroyImmediate
- 资源的优化建议
  - 1.预先制定一个详细和合理的美术规范
  - 2.在打assetbundle时管理资源之间的依赖
  - 3.加载资源的时候通过引用计数等方式管理资源的卸载
  - 4.压缩你的贴图 模型 动作,尤其是贴图(ETC /pvrtc !)
  - 5.gc策略
  - 6.小心代码部分的内存
  - 7.防止资源被长期占用
  - 8.善于使用分析工具 (profilor/android adb /ios instruments)
- 编辑器内部的资源使用
  - Assets/Editor Default Resources
  - EditorGUIUtility.Load