用法

将4个缓存相关文件导入项目，编译通过后在蓝图中搜索CacheDownLoadImage即可使用。由于未实现自动销毁功能，需要在程序关闭时执行ULocalCacheImage::Del()。

在LocalCacheImage.h的开头部分可以修改缓存配置。

若希望不通过CacheDownLoadImage，手动控制图片缓存与取用，可以直接获取缓冲池ULocalCacheImage::Get()，然后调用对应方法。

图片缓存池

基本设计思路

1. 使用AsyncSaveImage来缓存图片，使用AsyncUseImage或SyncUseMemCacheImage来加载图片。
2. 优先缓存到内存，内存满后将链表最前方的一半资源(最长时间未使用的资源)缓存到硬盘。缓存过程中可能存在本地io，故缓存与读取操作均为通过GraphTask异步执行。
3. 在本地缓存满了之后销毁旧数据的策略是移除链表最前方的资源(最长时间未使用的资源)
4. 为保证本地缓存在下次启动时依然生效，使用USaveGame类完成资产索引的保存与加载
5. 为避免个别存在的大型图片占满整个缓存，所以对大于ONEIMAGE_MAXSIZE的图片进行限制

问题或优化方向

1. 是否应该缓存图片时直接写入本地。当前遇到的问题是如果程序直接挂机，内存缓存将全部丢失
2. 这缓存思路应该是适用于任何资产的，应该存在更好的缓存策略
3. 缓存到硬盘的资源是否应该压缩，或者说是否可以扩展3级缓存
4. 是否有必要使用如此复杂的数据结构来保存资产，数据结构应该有很大的优化空间
5. 目前的保存方式是<url,图片>能否将url优化为hash值
6. 能否手动控制图片缓存优先级，或者通过简单的方法调整优先级策略

当前设计的具体实现方法

数据如何存储

1. 内存中的图片数据：

```
//使用USTRUCT主要是想用最简单的方法避免Texture被GC，此处有更优解
USTRUCT()
struct FMemImageData
{
    GENERATED_BODY()
public:
    //动态纹理2d直接记录纹理的指针时，使用起来是最快捷的，效率最高
    UPROPERTY()
    UTexture2DDynamic* Texture;

    //这里记录图片源数据，用于将纹理写入硬盘时使用。
    //此处有一个问题，纹理本身已经包含了一份原始数据，仅仅因为不便于取用就再保存一份有必要吗。
    TSharedPtr<TArray64<uint8>> TextureData;
```

```
    //请求图片时的url
    FString URL;

    float SizeX = 0;
    float SizeY = 0;
};
```

2. 硬盘中的图片数据：

```
/**
 * 在本地保存原始的图片数据，将会序列化到本地保存，这里只保留文件索引
 * 这里使用USTRUCT()是希望使用ue的序列化与反序列化机制
 */
USTRUCT()
struct FLocalImageData
{
    GENERATED_BODY()
public:
    //下载图片时使用的URL
    UPROPERTY()
    FString URL;

    //自相对路径起的文件名称,由用于构成文件名称的ID与类型后缀构成
    UPROPERTY()
    FString FileName;

    //文件大小 字节（Byte）
    UPROPERTY()
    uint32 size;

    //图片大小
    UPROPERTY()
    float SizeX = 0;
    UPROPERTY()
    float SizeY = 0;

    //上次使用时间戳
    UPROPERTY()
    int64 Timestamp;
};
```

3. 图片数据的索引与管理：

这里我使用3个容器来保存数据，分别用来随机查找、排序、序列化。这里查找与排序应该使用c++的std::map容器来替代这里两个容器，当时并未想到。

```
//本地缓存文件链表的索引，便于删除与移动链表节点
TMap<FString,TDoubleLinkedList<FString>::TDoubleLinkedListNode*>
LocalImageIndex;

//本地缓存文件索引，按照时间顺序排序的链表
TDoubleLinkedList<FString> LocalImageList;

//在启动时，需要将容器内的数据移动到链表中，维护在本地缓存的图片资源
UPROPERTY()
TMap<FString,FLocalImageData> LocalImageCache;
```

```
//本地缓存文件总大小 字节（Byte）
UPROPERTY()
int64 LocalAllSize;


//内存缓存文件链表的索引，利用Map存储，便于对链表的随机读写
TMap<FString,TDoubleLinkedList<FString>::TDoubleLinkedListNode*>
MemImageIndex;

//内存缓存文件索引，便于对图片资源按时间顺序排序
TDoubleLinkedList<FString> MemImageList;

//真正存储数据的Map，之所以没有直接使用链表或上方的map保存数据，主要是没有想到如何将上方的容器内的值添加到GC管理，故使用第3个map来保存纹理
UPROPERTY(Transient)
TMap<FString,FMemImageData> MemImageCache;

//内存缓存文件的总大小
int32 NowMemSize = 0;
```

如何向缓存中添加

1. 创建一个保存图片的异步任务，函数返回，静待任务开始执行。
2. 检查图片是否重复缓存、检查图片是否过大
3. 保存图片到内存中，并更新索引
4. 若内存缓冲区已满，则释放已播放内存缓存到硬盘
5. 若硬盘缓冲区已满，则删除最久未使用的资源

如何从缓存中取用

1. 提供一个直接读取内存缓存的方法，避免某些时候不希望异步加载。
2. 创建一个读取图片的异步任务，函数返回，静待任务开始执行。
3. 查找图片是否被缓存，是在哪里被缓存
4. 如果在内存被缓存，则更新内存排序并直接返回资源
5. 如果在硬盘被缓存，则读取文件，并更新索引
6. 解析文件成纹理资源，并保存到内存缓冲区。

如何处理异步逻辑

使用一个任务队列来避免所有时序问题，这也导致任务无法并行执行。

当出现任务时，根据ue自带的GraphTask系统创建一个异步task，在task中依次执行图片读取或保存的任务。

任务队列使用线程安全的TQueue容器，从而不必考虑锁相关问题。

图片下载

对DownLoadImage类扩展

为了使用户使用同DownLoadImage一致，不必考虑缓存问题，提供CacheDownLoadImage类。

修改了Activate函数：开启异步任务时会开始执行。将直接发起http请求改为先从缓冲区查找，未找到时才发起请求。

修改了HandleImageRequest函数：图片下载完成时会执行。当图片下载完成且成功时，将图片保存到缓冲区中。

开发过程中遇到另一个问题：若依次发起两个downloadimage请求，且第二个图片被缓存了，则必然导致先请求的图片由于网络延迟问题后返回结果。当然即时没有图片缓存功能这个问题依然存在。我对DownLoadImage节点添加一个额外的时间戳参数作为标志，用户只使用时间更晚的数据即可。