https://imzlp.com/posts/19135/

在开发游戏时，程序性能是需要着重考虑的问题，因为要尽可能覆盖最多的用户群体，就要考虑那些中低端设备的运行效果，兼容非常多配置差异的硬件，在这种情况下，怎么样分析和优化游戏的性能瓶颈是关键。

在运行时把更多的资源加载至内存中，本质上是一种空间换时间的思路。因为频繁从磁盘进行 IO 是非常耗时的，把资源预先加载到内存就可以实现高速读取，但是内存资源也是有限的，并不能不加限制地使用，尤其是对某些中低端移动设备而言，4G 甚至更小的内存的设备目前还具有不少的占有率，所以在内存方面不能浪费，而且过高的内存占用也有可能导致被系统查杀。

内存优化本质上就是在加载效率和内存占用之间寻求一个平衡，怎么样在能满足兼容更多低配设备正常运行不触发 OOM 的同时尽可能地把可利用的内存使用起来，提高程序的运行效率。

准备写几篇性能优化相关的文章，本篇文章先从 UE 内存分析入手，介绍常用的内存分析工具和方法，以及对 UE 项目中能够进行的内存优化手段做一个整理，这部分内容之前以笔记的形式记录在 notes/ue 中，后续内存相关的内容都会补充到本篇文章。

内存优化其实主要就是从以下四个方面着手：

1. 排查内存泄漏
2. 裁剪多余模块
3. 优化现有模块的内存占用
4. 有损优化：砍内容（素材质量等）

三步走：查 bug、挤水分、砍需求。

内存分析工具

所以，在进行内存优化之前，首先要能够对 UE 项目的内存分布有一个大概的了解，可以使用 UE 提供的内存分析工具以及一些 Native 平台的分析工具。

内存分析资料：

- LLM
- UE4 内存 Profiler
- Android 平台上的 Native 内存分析

一些常用的 console command:

```
stat memory #显示引擎中各个子系统的内存占用
stat MemoryAllocator #显示内存分配信息
stat MemoryPlatform #显示平台内存信息
stat MemoryStaticMesh #显示静态模型的内存信息
```

以及开启 LLM，在启动时加上 `-LLM` 参数

```
-LLM #启用LLM
-LLMCSV #连续将所有值写入CSV文件。自动启用-LLM。
-llmtagsets=Assets #实验性功能。显示每个资源分配的内存总计。
-llmtagsets=AssetClasses #实验性功能。显示每个UObject类类型的总计。
```

然后就可以在运行时使用以下 console 命令：

```
stat llm #显示LLM摘要。所有较低级别的引擎统计信息都归入单个引擎统计信息。
stat llmfull #显示LLM所有统计信息
stat LLMPlatform #显示从OS分配的所有内存信息
stat LLMOverhead #显示LLM内部使用的内存
```

内存分析还可以使用以下工具：

- memreport
- MemoryProfiler
- Heapprofd(Android)
- Instrument(IOS)

在游戏的 console 中输入 `memreport`(-full) 会在 `Saved/Profiling/Memreports` 中创建地图目录以及 `.memreport` 文件，可以使用文本编辑器打开，能够看到游戏中各个部分的内存占用情况。

具体的内存分析工具的使用和对 UE 引擎中内存分配的分析流程有时间再详细补充，关于 LLM 的内容可以详情参考 UE 的文档。

内存优化方案

以下列举的优化方式，其实都是可选的，并不是一定要把所有的都做了就最好，因为内存优化要兼顾效率，所以可以根据项目需求在不同的设备上控制要优化的功能，尽可能再保证功能一致地情况下对低端机进行适配。

这里主要列举 UE 中哪些部分可以被优化以及如果做，具体的优化数据有时间慢慢分析和补充。

关闭不必要的功能支持

根据需求可以裁剪以下的引擎模块支持：

- APEX：如果不使用 Nvidia 的 APEX 破碎系统，可以在编译引擎时去掉 APEX 的支持。可以在 BuildSetting 或者 TargetRules 设置 `bCompileAPEX=true`。
- Recast (NavMesh)：如果客户端在运行时不需要 Recast 的支持，并且不需要客户端本地进行 NavMesh 寻路操作，可以运行时裁剪掉 NavMesh 的支持。可以在 BuildSetting 或者 TargetRules 设置 `bCompileRecast=true`。
- FreeType：是否需要 FreeType 字库支持，可以在 BuildSetting 或者 TargetRules 设置 `bCompileFreeType=true`。
- ICU (unicode/i18n)：引擎 Core 模块中对 unicode/i18n 的支持，可以在 BuildSetting 或者 TargetRules 设置 `bCompileICU=true`。
- CompileForSize：UE 提供的优化选项，可以控制编译时严格控制大小，但是会牺牲性能。可以在 BuildSetting 或者 TargetRules 设置 `bCompileForSize=true`。
- CEF3：可选是否支持 `Chromium Embedded Framework`，Google 的嵌入式浏览器支持。可以在 BuildSetting 或者 TargetRules 设置 `bCompileCEF3=true`。
- bUsesSteam：是否使用 Steam，手游可以关闭，在 TargetRules 中通过 `bUsesSteam` 控制。
- SpeedTree：如果游戏中不需要使用 SpeedTree 进行植被建模，可以关闭编译 SpeedTree，通过 TargetRules 中的 `bOverrideCompileSpeedTree` 控制。

- Audio 模块：如果项目使用 WWise 等作为音频播放接口，如果完全不需要引擎中内置的 Audio 模块，该部分功能是冗余的，可以裁剪掉。
- 国际化模块：如果游戏的多语言支持不依赖 UE 的文本采集和翻译功能，可以裁剪掉该模块。

可以减少编译之后静态程序的大小以及减少不必要的执行逻辑。

控制 AssetRegistry 的序列化

AssetRegistry 其实主要是在 Editor 下用来方便进行资源的查找和过滤操作，它的主要使用者是 ContentBrowser，这一点在 UE 的文档中也有描述：Asset Registry。

对于项目而言在 Runtime 可能没有需求来使用它，但是在 `AssetRegistry` 模块一启动就会把 `AssetRegistry.bin` 加载到内存中，如果对它没有需求其实这部分内存是浪费的。

好在 UE 提供了不序列化或者部分序列化 AssetRegistry 数据的方法，在 `UAssetRegistryImpl` 的构造函数中会调用 `InitializeSerializationOptionsFromIni` 函数来读取 `DefaultEngine.ini` 中的配置，并会构造出一个 `FAssetRegistrySerializationOptions` 结构来存储，它会在后续的 `Serialize` 函数中使用，用来控制把哪部分的数据序列化到 `AssetRegistry` 中。

```
Runtime/AssetRegistry/Private/AssetRegistry.cpp
void
UAssetRegistryImpl::InitializeSerializationOptionsFromIni(FAssetRegistrySerializ
ationOptions& Options, const FString& PlatformIniName) const
{
  FConfigFile* EngineIni = nullptr;
#if WITH_EDITOR
  // Use passed in platform, or current platform if empty
  FConfigFile PlatformEngineIni;
  FConfigCacheIni::LoadLocalIniFile(PlatformEngineIni, TEXT("Engine"), true,
(!PlatformIniName.IsEmpty() ? *PlatformIniName :
ANSI_TO_TCHAR(FPlatformProperties::IniPlatformName())));
  EngineIni = &PlatformEngineIni;
#else
  // In cooked builds, always use the normal engine INI
  EngineIni = GConfig->FindConfigFile(GEngineIni);
#endif

  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializeAssetRegistry"),
Options.bSerializeAssetRegistry);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializeDependencies"),
Options.bSerializeDependencies);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializeNameDependencies"),
Options.bSerializeSearchableNameDependencies);
  EngineIni->GetBool(TEXT("AssetRegistry"),
TEXT("bSerializeManageDependencies"), Options.bSerializeManageDependencies);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializePackageData"),
Options.bSerializePackageData);
  EngineIni->GetBool(TEXT("AssetRegistry"),
TEXT("bUseAssetRegistryTagsWhitelistInsteadOfBlacklist"),
Options.bUseAssetRegistryTagsWhitelistInsteadOfBlacklist);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bFilterAssetDataWithNoTags"),
Options.bFilterAssetDataWithNoTags);
  EngineIni->GetBool(TEXT("AssetRegistry"),
TEXT("bFilterDependenciesWithNoTags"), Options.bFilterDependenciesWithNoTags);
```

```
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bFilterSearchableNames"),
Options.bFilterSearchableNames);
  // ...
}
```

这个控制方式可以在打包时控制是否生成 AssetRegistry.bin，以及控制在运行时反序列化哪些 AssetRegistry 的数据（但是不会对 DevelopmentAssetRegistry.bin 造成影响，可以用它来进行资产审计）。

它的反序列化流程为：

1. 检测 `bSerializeAssetRegistry`，如果为 `true` 则把 AssetRegistry.bin 以二进制形式加载到内存中
2. 通过 `Serialize` 函数来把二进制数据反序列化
3. 释放加载 AssetRegistry.bin 所占用的内存

所以，AssetRegistry 的内存占用是在序列化之后的数据，而 `FAssetRegistrySerializationOptions` 就是控制把哪些数据序列化的。

```
Runtime/AssetRegistry/Public/AssetRegistryState.h
/** Load/Save options used to modify how the cache is serialized. These are read
out of the AssetRegistry section of Engine.ini and can be changed per platform.
*/
struct FAssetRegistrySerializationOptions
{
  /** True rather to load/save registry at all */
  bool bSerializeAssetRegistry;

  /** True rather to load/save dependency info. If true this will handle hard
and soft package references */
  bool bSerializeDependencies;

  /** True rather to load/save dependency info for Name references,  */
  bool bSerializeSearchableNameDependencies;

  /** True rather to load/save dependency info for Manage references,  */
  bool bSerializeManageDependencies;

  /** If true will read/write FAssetPackageData */
  bool bSerializePackageData;

  /** True if CookFilterlistTagsByClass is a whitelist. False if it is a
blacklist. */
  bool bUseAssetRegistryTagsWhitelistInsteadOfBlacklist;

  /** True if we want to only write out asset data if it has valid tags. This
saves memory by not saving data for things like textures */
  bool bFilterAssetDataWithNoTags;

  /** True if we also want to filter out dependency data for assets that have no
tags. Only filters if bFilterAssetDataWithNoTags is also true */
  bool bFilterDependenciesWithNoTags;

  /** Filter out searchable names from dependency data */
  bool bFilterSearchableNames;
```

```
  /** The map of classname to tag set of tags that are allowed in cooked builds.
This is either a whitelist or blacklist depending on
bUseAssetRegistryTagsWhitelistInsteadOfBlacklist */
  TMap<FName, TSet<FName>> CookFilterlistTagsByClass;

  FAssetRegistrySerializationOptions()
    : bSerializeAssetRegistry(false)
    , bSerializeDependencies(false)
    , bSerializeSearchableNameDependencies(false)
    , bSerializeManageDependencies(false)
    , bSerializePackageData(false)
    , bUseAssetRegistryTagsWhitelistInsteadOfBlacklist(false)
    , bFilterAssetDataWithNoTags(false)
    , bFilterDependenciesWithNoTags(false)
    , bFilterSearchableNames(false)
  {}

  /** Options used to read/write the DevelopmentAssetRegistry, which includes
all data */
  void ModifyForDevelopment()
  {
    bSerializeAssetRegistry = bSerializeDependencies =
bSerializeSearchableNameDependencies = bSerializeManageDependencies =
bSerializePackageData = true;
    DisableFilters();
  }

  /** Disable all filters */
  void DisableFilters()
  {
    bFilterAssetDataWithNoTags = false;
    bFilterDependenciesWithNoTags = false;
    bFilterSearchableNames = false;
  }
};
```

配置的读取在以下代码中：

```
Runtime/AssetRegistry/Private/AssetRegistry.cpp
void
UAssetRegistryImpl::InitializeSerializationOptionsFromIni(FAssetRegistrySerializ
ationOptions& Options, const FString& PlatformIniName) const
{
  FConfigFile* EngineIni = nullptr;
#if WITH_EDITOR
  // Use passed in platform, or current platform if empty
  FConfigFile PlatformEngineIni;
  FConfigCacheIni::LoadLocalIniFile(PlatformEngineIni, TEXT("Engine"), true,
(!PlatformIniName.IsEmpty() ? *PlatformIniName :
ANSI_TO_TCHAR(FPlatformProperties::IniPlatformName())));
  EngineIni = &PlatformEngineIni;
#else
  // In cooked builds, always use the normal engine INI
  EngineIni = GConfig->FindConfigFile(GEngineIni);
```

```
#endif

  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializeAssetRegistry"),
Options.bSerializeAssetRegistry);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializeDependencies"),
Options.bSerializeDependencies);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializeNameDependencies"),
Options.bSerializeSearchableNameDependencies);
  EngineIni->GetBool(TEXT("AssetRegistry"),
TEXT("bSerializeManageDependencies"), Options.bSerializeManageDependencies);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bSerializePackageData"),
Options.bSerializePackageData);
  EngineIni->GetBool(TEXT("AssetRegistry"),
TEXT("bUseAssetRegistryTagsWhitelistInsteadOfBlacklist"),
Options.bUseAssetRegistryTagsWhitelistInsteadOfBlacklist);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bFilterAssetDataWithNoTags"),
Options.bFilterAssetDataWithNoTags);
  EngineIni->GetBool(TEXT("AssetRegistry"),
TEXT("bFilterDependenciesWithNoTags"), Options.bFilterDependenciesWithNoTags);
  EngineIni->GetBool(TEXT("AssetRegistry"), TEXT("bFilterSearchableNames"),
Options.bFilterSearchableNames);

  TArray<FString> FilterlistItems;
  if (Options.bUseAssetRegistryTagsWhitelistInsteadOfBlacklist)
  {
    EngineIni->GetArray(TEXT("AssetRegistry"), TEXT("CookedTagsWhitelist"),
FilterlistItems);
  }
  else
  {
    EngineIni->GetArray(TEXT("AssetRegistry"), TEXT("CookedTagsBlacklist"),
FilterlistItems);
  }

  {
    // this only needs to be done once, and only on builds using
USE_COMPACT_ASSET_REGISTRY
    TArray<FString> AsFName;
    EngineIni->GetArray(TEXT("AssetRegistry"), TEXT("CookedTagsAsFName"),
AsFName);
    TArray<FString> AsPathName;
    EngineIni->GetArray(TEXT("AssetRegistry"), TEXT("CookedTagsAsPathName"),
AsPathName);
    TArray<FString> AsLocText;
    EngineIni->GetArray(TEXT("AssetRegistry"), TEXT("CookedTagsAsLocText"),
AsLocText);
    FAssetRegistryState::IngestIniSettingsForCompact(AsFName, AsPathName,
AsLocText);
  }
  // ...

}
```

在 `Config/DefaultEngine.ini` 中创建 `AssetRegistry` Section 使用上面的名字就可以控制 AssetRegistry 的序列化，减少打包时的包体大小以及内存占用（AssetRegistry 在引擎启动时会加载到内存中）

```
DefaultEngine.ini
[AssetRegistry]
bSerializeAssetRegistry=false
bSerializeDependencies=false
bSerializeNameDependencies=false
bSerializeManageDependencies=false
bSerializePackageData=false
```

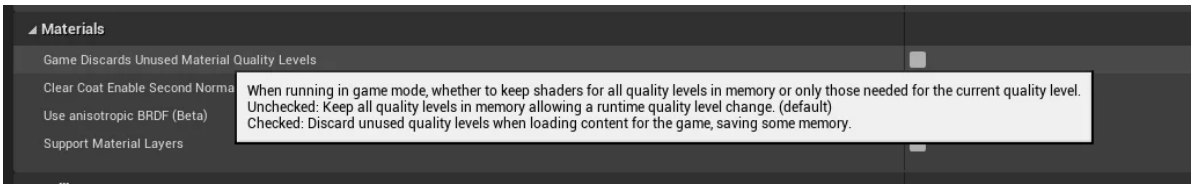
也可以对某个平台来单独指定，只需要修改平台相关的 Ini 文件：

```
Config/Windows/WindowsEngine.ini
Config/Android/AndroidEngine.ini
Config/IOS/IOSEngine.ini
```

只加载所使用质量级别的 Shader

默认情况下，引擎会把所有质量级别的 Shader 加载到内存中，在不需要实施切换画质的情况下，可以不加载未使用的质量级别，降低 Shader 的内存占用。

在 `Project Settings`-`Engine`-`Rendering`-`Materials`-`Game Discards Unused Material Quality Levels`：



或者在 `DefaultEngine.ini` 中添加以下配置：

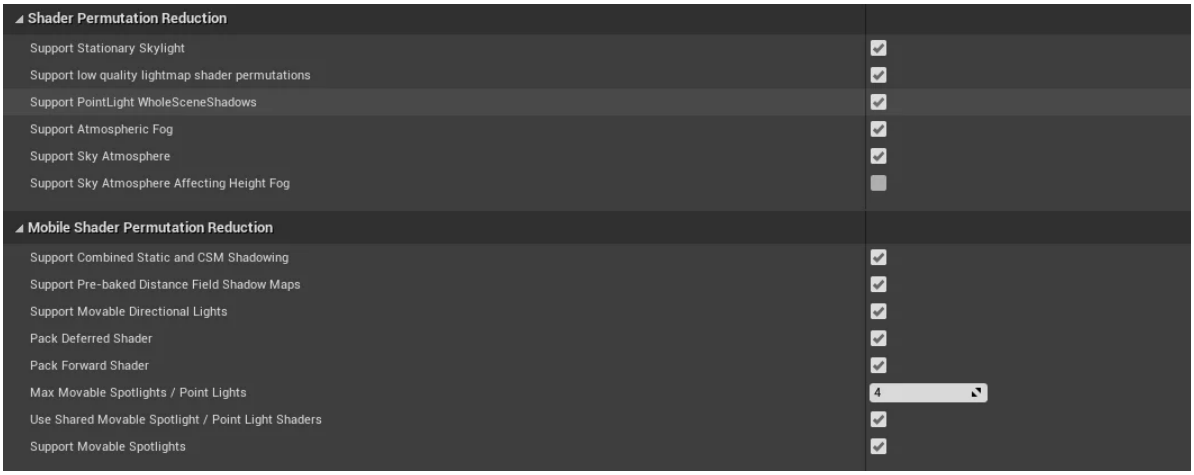
```
DefaultEngine.ini
[/Script/Engine.RendererSettings]
r.DiscardUnusedQuality=True
```

> When running in game mode, whether to keep shaders for all quality levels in memory or only those needed for the current quality level.

- Unchecked: Keep all quality levels in memory allowing a runtime quality level change. (default)
- Checked: Discard unused quality levels when loading content for the game, saving some memory.

减少 Shader 变体

可以通过减少母材制的数量以及在 `Project Settings`-`Engine`-`Rendering` 中开启下列选项：



ShareMaterialShaderCode

在打包时可以在 `Project Settings`-`Packaging` 中设置 `Share Material Shader Code` 和 `Shadred Material Native Libraries` 可以减小包体的大小，并且会减少内存占用（增加加载时间）。

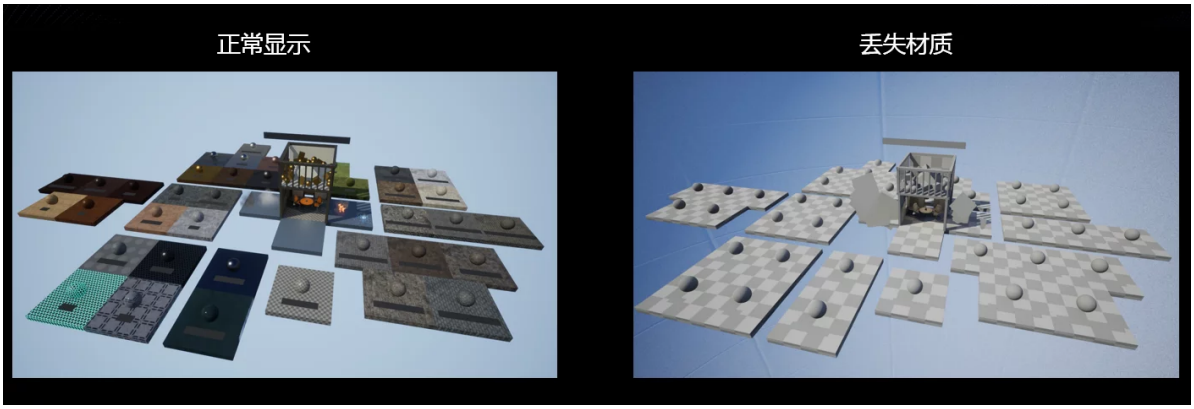
```
/**
 * By default shader code gets saved inline inside material assets,
 * enabling this option will store only shader code once as individual files
 * This will reduce overall package size but might increase loading time
 */
UPROPERTY(config, EditAnywhere, Category=Packaging)
bool bShareMaterialShaderCode;

/**
 * By default shader shader code gets saved into individual platform agnostic files,
 * enabling this option will use the platform-specific library format if and only if one is available
 * This will reduce overall package size but might increase loading time
 */
UPROPERTY(config, EditAnywhere, Category=Packaging, meta = (EditCondition = "bShareMaterialShaderCode", ConfigRestartRequired = true))
bool bSharedMaterialNativeLibraries;
```

开启了之后打出的包中会生成下列文件：

```
ShaderArchive-Blank425-PCD3D_SM5.ushaderbytecode
ShaderCode-Global-PCD3D_SM5.ushaderbytecode
```

但是，如果开启之后如果后续的 Cook 资源 Shader 发生了变动，而基础包内还是旧的 ShaderBytecode 信息，会导致材质丢失。



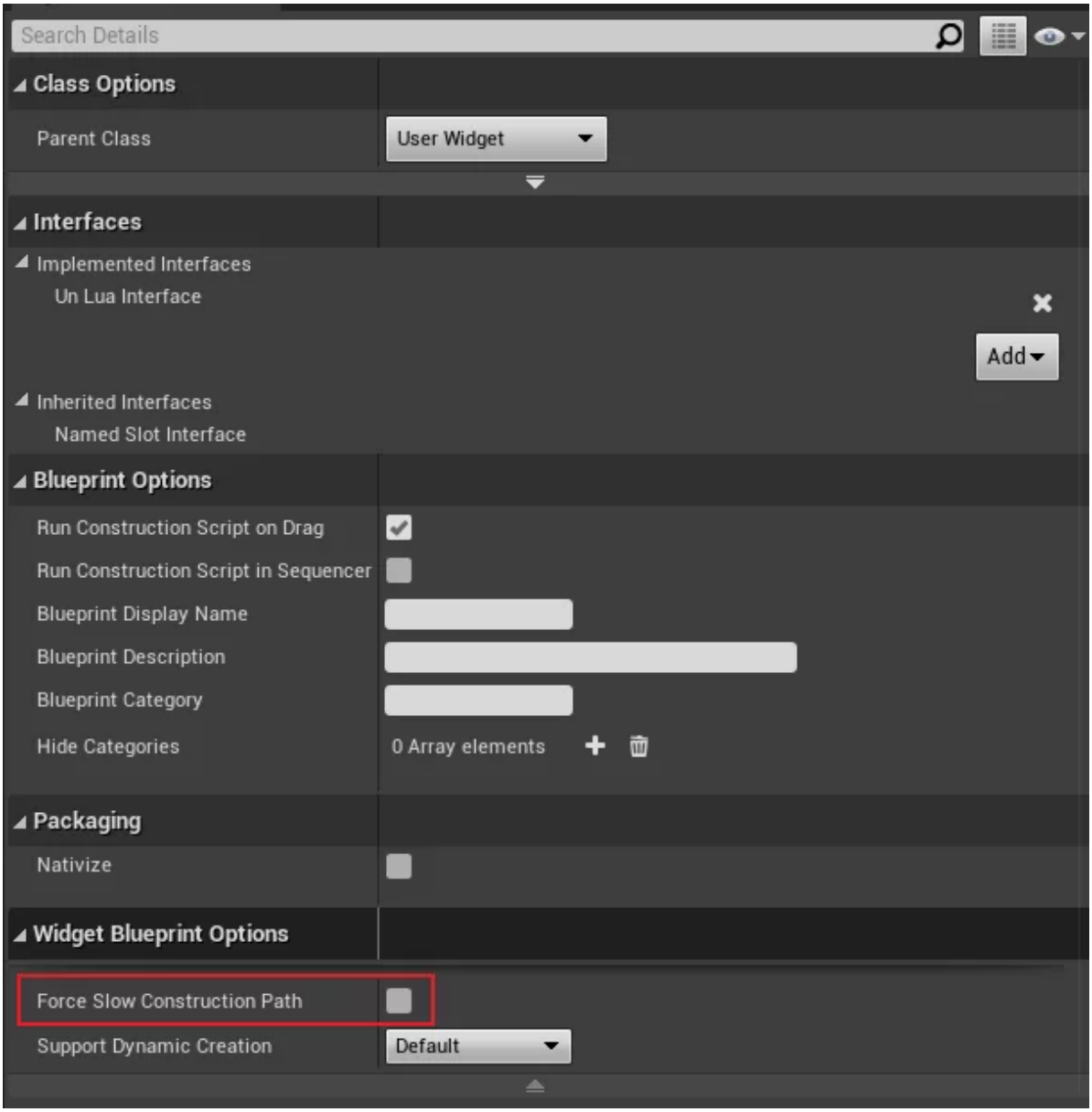
有三个办法：

1. 后续的打包时可以把 Shaderbytecode 文件打包在 pak 中，挂载时加载；
2. Cook 热更资源时把 Shaderbytecode 打包在资源内；
3. 创建 ShaderPatch，在热更后加载；

热更 Shaderbytecode 更具体地实践流程可以在我之前对我文章 UE4 热更新：Create Shader Patch 中查看。

关闭 UMG 的模板化创建

引擎中有缓存蓝图控件加速创建的功能，但是会造成内存的浪费，可以配置关闭：



也可以直接修改引擎中的代码使用类内初始化给予默认值：

```
Source\Editor\UMGEditor\Public\WidgetBlueprint.h
UPROPERTY(EditAnywhere, AdvancedDisplay, Category=WidgetBlueprintOptions,
AssetRegistrySearchable)
bool bForceSlowConstructionPath;
```

该变量在以下代码中被检测使用：

```
Source\Editor\UMGEditor\Private\WidgetBlueprintCompiler.cpp
bool FWidgetBlueprintCompilerContext::CanAllowTemplate(FCompilerResultsLog&
MessageLog, UWidgetBlueprintGeneratedClass* InClass)
{
  // ...
  // If this widget forces the slow construction path, we can't template it.
  if ( WidgetBP->bForceSlowConstructionPath )
  {
    if (GetDefault<UUMGEditorProjectSettings>()-
>CompilerOption_CookSlowConstructionWidgetTree(WidgetBP))
    {
      MessageLog.Note(*LOCTEXT("ForceSlowConstruction", "Fast Templating
Disabled By User.").ToString());
```

```
      return false;
    }
    else
    {
      MessageLog.Error(*LOCTEXT("UnableToForceSlowConstruction", "This project has [Cook Slow Construction Widget Tree] disabled, so [Force Slow Construction Path] is no longer allowed.").ToString());
    }
  }
  // ...
}
```

关闭 pakcache

引擎中默认启用了 PakCache 机制，在从 Pak 中读取文件时，会多读一段内存用作缓存，内存占用还是十分可观的（通过 `stat memory` 查看）：

游戏启动时会有 PakCache 的 Log：

```
[2021.03.23-10.49.21:354][445]LogPakFile: Precache HighWater 16MB
[2021.03.23-10.49.21:382][447]LogPakFile: Precache HighWater 32MB
[2021.03.23-10.49.21:442][450]LogPakFile: Precache HighWater 48MB
[2021.03.23-10.49.21:470][452]LogPakFile: Precache HighWater 64MB
```

可以通过以下方式配置关闭：

```
[ConsoleVariables]
pakcache.Enable=0
```

关闭 PakCache 会带来频繁 IO 的问题，但是具体的性能影响细节要等有时间再来分析。

Unload pakentry filenames

从 UE4.23 开始，引擎中提供了 Mount PakFile 的内存优化配置：

```
DefaultEngine.ini
[Pak]
UnloadPakEntryFilenamesIfPossible=true
DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames="*/Config/Tags/"
+DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames="*/Content/Localization/*"
ShrinkPakEntriesMemoryUsage=true
```

在 FPakPlatformFile 执行 Initialize 的时候会绑定 `FCoreDelegates::OnOptimizeMemoryUsageForMountedPaks`，可以调用该 Delegate 来通知 PakPlatformFile 来优化已 mount 的 Pak 的内存。

```
Runtime\PakFile\Private\IPlatformFilePak.cpp
void FPakPlatformFile::OptimizeMemoryUsageForMountedPaks()
{
#if !(IS_PROGRAM || WITH_EDITOR)
    FSlowHeartBeatScope SuspendHeartBeat;
    bool bUnloadPakEntryFilenamesIfPossible = FParse::Param(FCommandLine::Get(),
TEXT("unloadpakentryfilenames"));
    GConfig->GetBool(TEXT("Pak"), TEXT("UnloadPakEntryFilenamesIfPossible"),
bUnloadPakEntryFilenamesIfPossible, GEngineIni);

    if ((bUnloadPakEntryFilenamesIfPossible &&
!FParse::Param(FCommandLine::Get(), TEXT("nounloadpakentries"))) ||
FParse::Param(FCommandLine::Get(), TEXT("unloadpakentries")))
    {
        // With [Pak] UnloadPakEntryFilenamesIfPossible enabled, [Pak]
DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames
        // can contain pak entry directory wildcards of which the entire
recursive directory structure of filenames underneath a
        // matching wildcard will be kept.
        //
        // Example:
        //   [Pak]
        //
DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames="*/Config/Tags/"
        //
+DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames="*/Content/Localizat
ion/*"
        TArray<FString> DirectoryRootsToKeep;
        GConfig->GetArray(TEXT("Pak"),
TEXT("DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames"),
DirectoryRootsToKeep, GEngineIni);

        FPakPlatformFile* PakPlatformFile = (FPakPlatformFile*)
(FPlatformFileManager::Get().FindPlatformFile(FPakPlatformFile::GetTypeName()));
        PakPlatformFile->UnloadPakEntryFilenames(&DirectoryRootsToKeep);
    }

    bool bShrinkPakEntriesMemoryUsage = FParse::Param(FCommandLine::Get(),
TEXT("shrinkpakentries"));
    GConfig->GetBool(TEXT("Pak"), TEXT("ShrinkPakEntriesMemoryUsage"),
bShrinkPakEntriesMemoryUsage, GEngineIni);
    if (bShrinkPakEntriesMemoryUsage)
    {
        FPakPlatformFile* PakPlatformFile = (FPakPlatformFile*)
(FPlatformFileManager::Get().FindPlatformFile(FPakPlatformFile::GetTypeName()));
        PakPlatformFile->ShrinkPakEntriesMemoryUsage();
    }
#endif
}
```

- UnloadPakEntryFilenamesIfPossible：允许卸载 PakEntry filenames 占用的内存

- DirectoryRootsToKeepInMemoryWhenUnloadingPakEntryFilenames：卸载 PakEntry filename 时要保留的目录
- bShrinkPakEntriesMemoryUsage：缩小 PakEntry 的内存占用

当调用之后，如果开启 `UnloadPakEntryFilenamesIfPossible` 了，会通过计算 Pak 中文件名列表的 Hash 来节省内存，但是卸载 PakEntry filenames 之后无法再使用路径的通配符匹配。

```
Runtime\PakFile\Public\IPlatformFilePak.h
/** Iterator class used to iterate over all files in pak. */
class FFileIterator
{
    // ...
    /**
     * Saves memory by hashing the filenames, if possible. After this process,
     * wildcard scanning of pak entries can no longer be performed. Returns TRUE
     * if the process successfully unloaded filenames from this pak
     *
     * @param CrossPakCollisionChecker A map of hash->fileentry records
encountered during filename unloading on other pak files. Used to detect
collisions with entries in other pak files.
     * @param DirectoryRootsToKeep An array of strings in wildcard format that
specify whole directory structures of filenames to keep in memory for directory
iteration to work.
     * @param bAllowRetries If a collision is encountered, change the intial
seed and try again a fixed number of times before failing
     */
    bool UnloadPakEntryFilenames(TMap<uint64, FPakEntry>&
CrossPakCollisionChecker, TArray<FString>* DirectoryRootsToKeep = nullptr, bool
bAllowRetries = true);
};
```

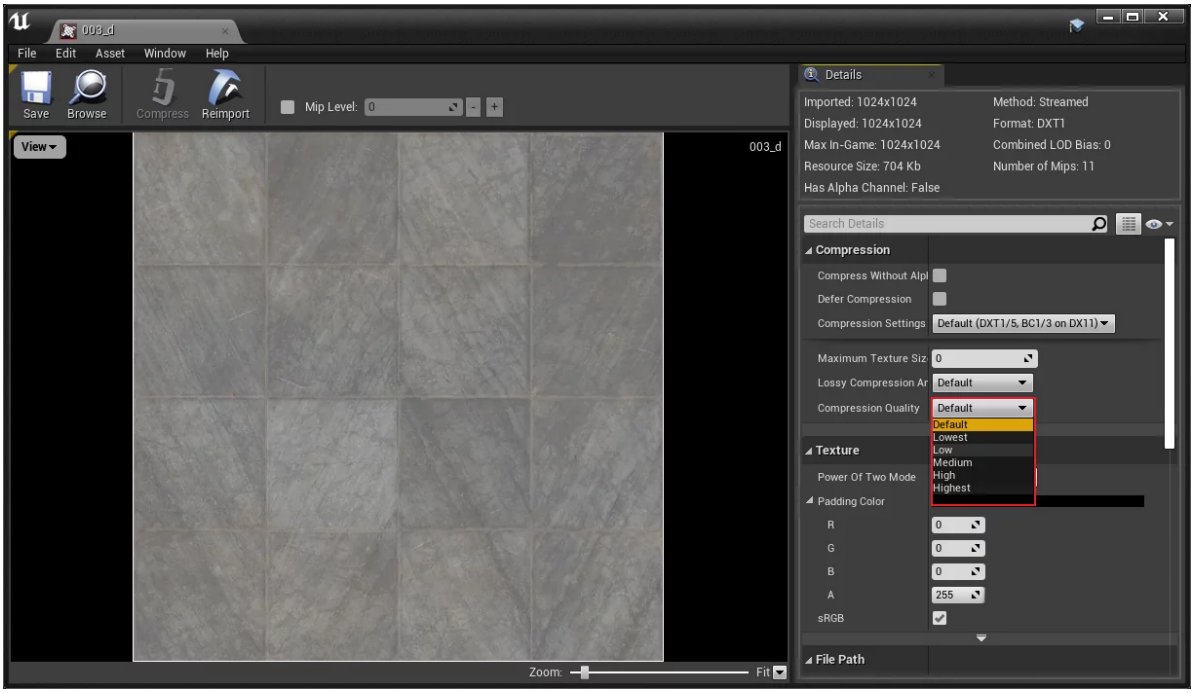
压缩 Texture

Texture 的压缩是有损压缩，能够减小包体大小以及加载到内存中的大小，虽然是有损压缩，但是在移动端质量降低的效果并不明显，可以根据项目情况进行设置。

之前的笔记中，提到过可以在 `Project Settings-Cooker-Texture-ASTC Compression vs Size` 可以设置默认的资源质量和大小的级别：

```
0=12x12
1=10x10
2=8x8
3=6x6
4=4x4
```

在 Texture 的资源编辑中也可以针对某个 Texture 单独设置：



Lowest->Hightest 对应着 `0-4` 的值，使用 Default 则使用项目设置中的配置。

并且，设置 `Compression Settings` 的类型也会对资源压缩的类型有差别，Default 则是项目设置中的参数，如果设置成 NormalMap 的类型会是 `ASTC_4x4` 的。

- Using ASTC Texture Compression for Game Assets
- Texture Compression Settings

UPDATE

使用 Github Gist 管理的动态更新内容，在国内网络可能会无法查看。