# UE基础设施

## 系统配置

为了解析UE引擎架构，本书以开源的UE代码为基础，把UE引擎中的核心代码抽出来，做成一个单独的引擎，以便于分析UE引擎部分的代码，为了方便描述暂且叫它YiEngine，Yi，易也，有简易、变易和不易之意，以抽丝剥茧的方式，庖丁解牛的手段深入浅析地分析UE引擎代码。

本系统用CMake编译，首先建立根文件夹YiEngine，在YiEngine中建议以下文件夹Core和Yi，和文件CMakeLists.txt，CMakeLists.txt的主要内容如下：

cmake\_minimum\_required(VERSION 3.8.0)

set(CMAKE\_ALLOW\_LOOSE\_LOOP\_CONSTRUCTS TRUE)

cmake\_policy(SET CMP0003 NEW)

指定CMake的最新版本。

project(YiEngine)

指定工程名，如果用VC编译的话，Solution的名字为YiEngine.sln。

# Specify build paths

set(CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY "${YiEngine\_BINARY\_DIR}/lib")

set(CMAKE\_LIBRARY\_OUTPUT\_DIRECTORY "${YiEngine\_BINARY\_DIR}/lib")

set(CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY "${YiEngine\_BINARY\_DIR}/bin")

指定可执行文件和链接库的输出路径

add\_subdirectory(Core)

add\_subdirectory(Yi)

添加子文件，以搜索这些子文件中的CMakeLists.txt。

在Core文件夹中包含以下内容：GenericPlatform，HAL，Linux，Unix，Windows文件夹和CMakeLists.txt文件。

# 项目名称

project(Core)

# 源代码文件

file(GLOB\_RECURSE project\_headers \*.h)

file(GLOB\_RECURSE project\_cpps \*.cpp)

set(${PROJECT\_NAME}\_SOURCE ${project\_headers} ${project\_cpps})

source\_group\_by\_dir(${PROJECT\_NAME}\_SOURCE)

# library

add\_library(${PROJECT\_NAME} ${${PROJECT\_NAME}\_SOURCE})

Yi文件下CMakeLists.txt主要内容

# 项目名称

project(Yi)

# 源代码文件

file(GLOB\_RECURSE project\_headers \*.h)

file(GLOB\_RECURSE project\_cpps \*.cpp)

set(${PROJECT\_NAME}\_SOURCE ${project\_headers} ${project\_cpps})

source\_group\_by\_dir(${PROJECT\_NAME}\_SOURCE)

# 头文件搜索目录

include\_directories(

${YiEngine\_SOURCE\_DIR}/Core

)

# EXE

add\_executable(${PROJECT\_NAME} WIN32 ${${PROJECT\_NAME}\_SOURCE})

# 链接的库

set(ALL\_LIBS

Core.lib

)

# 链接库搜索目录

target\_link\_directories(${PROJECT\_NAME} PUBLIC ${CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY})

target\_link\_libraries(${PROJECT\_NAME} ${ALL\_LIBS})

add\_dependencies(${PROJECT\_NAME} Core)

## 跨平台架构

UE支持多种平台。

目录结构

Core

GenericPlatform

HAL

Windows

Linux

GenericPlatform定义各种基类，包括线程等，放置各种平台通用的定义和实现。Windows和Linux放置平台相关的实现，HAL是平台无关的接口。

## 类型定义

首先定义在Core/ GenericPlatform/ GenericPlatform.h定义基类FGenericPlatformTypes，该类把各种平台通用的类型重新定义为新的符号，比如unsigned char定义为uint8.

/\*\*

 \* Generic types for almost all compilers and platforms

 \*\*/

struct FGenericPlatformTypes {

// Unsigned base types.

// 8-bit  unsigned.

    typedef unsigned char      uint8;

};

在Windows/WIndowsPlatform.h头文件中定义继承FGenericPlatformTypes的类FWindowsPlatformTypes，该类定义针对Windows平台的实现，比如SIZE\_T。并用以下重定义语句把FWindowsPlatformTypes定义为FPlatformTypes，在使用过程中只会引用FPlatformTypes，以引用不同平台的定义。

/\*\*

\* Windows specific types

\*\*/

struct FWindowsPlatformTypes : public FGenericPlatformTypes

{

#ifdef \_WIN64

typedef unsigned \_\_int64 SIZE\_T;

typedef \_\_int64 SSIZE\_T;

#else

typedef unsigned long SIZE\_T;

typedef long SSIZE\_T;

#endif

};

typedef FWindowsPlatformTypes FPlatformTypes;

在HAL/Platform.h中，根据预定义的平台相关的宏来包含不同平台的头文件，比如如果PLATFORM\_WINDOWS定义为1，那么只包含WindowsPlatform.h，如果PLATFORM\_LINUX定义为1，那么只包含LinuxPlatform.h，当使用类型定义FPlatformTypes::SIZT\_T时，根据

#ifdef \_MSC\_VER

#define PLATFORM\_WINDOWS 1

#else

#define PLATFORM\_LINUX 1

#endif

#include "../GenericPlatform/GenericPlatform.h"

#if PLATFORM\_WINDOWS

#include "../Windows/WindowsPlatform.h"

#endif

#if PLATFORM\_LINUX

#include "../Linux/LinuxPlatform.h"

#endif

这样使用类型定义比较麻烦，在HAL/Platform.h文件中，把平台相关的类型转化为全局定义：

//------------------------------------------------------------------

// Transfer the platform types to global types

//------------------------------------------------------------------

//~ Unsigned base types.

/// An 8-bit unsigned integer.

typedef FPlatformTypes::uint8 uint8;