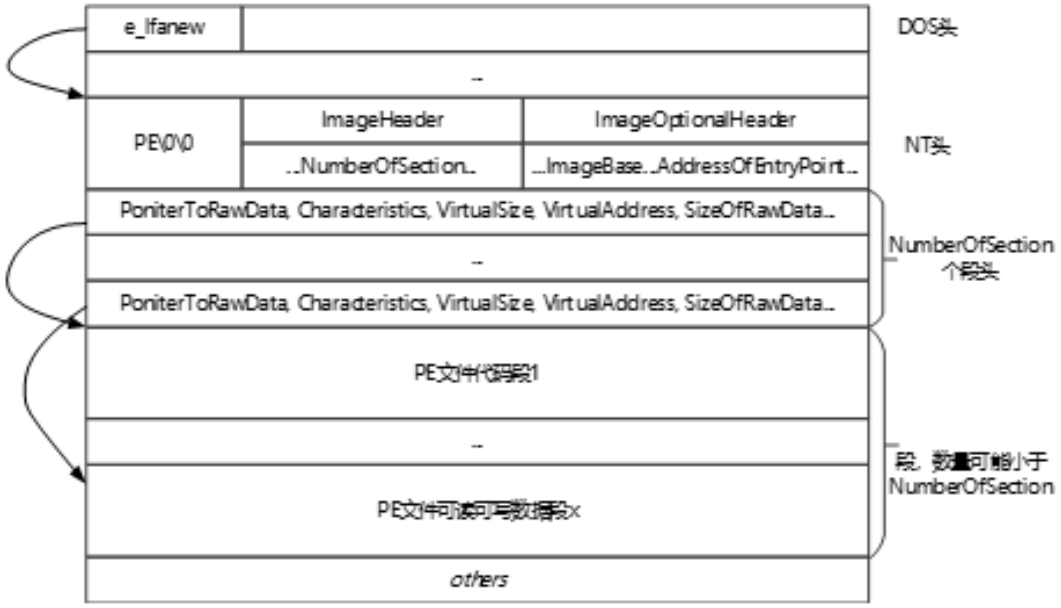
Unix V6++ 系统，可执行文件格式解析

同济大学计算机系操作系统　　邓蓉

可执行文件格式解析 PEParser.h & PEParser.cpp

Unix V6++系统可识别的可执行文件是 PE 格式的，标识以 magic number 0x00004550（字符串 PE\0\0）。下图是 PE 可执行文件的格式，与应用程序装载相关的是可执行文件头部登记的 3 个数据结构和初始化逻辑段所需的数据。这 3 个数据结构分别是：DOS 头，NT 头和段表。随后依次排列应用程序的代码段（应用程序需要调用的所有子程序），数据段（带初值的全局变量的初值）和 只读数据段（常数的值）。



一、DOS 头

Unix V6++系统，定义 DOS 头的数据结构是 ImageDosHeader，我们只用它的 e_lfanew 字段。该字段的值是 0x3Ch，表示随后的 NT 头，文件偏移量是 0x3Ch 字节。


```
struct ImageDosHeader
{                                               // DOS .EXE header
    unsigned short   e_magic;                   // Magic number
    unsigned short   e_cblp;                    // Bytes on last page of file
    unsigned short   e_cp;                      // Pages in file
    unsigned short   e_crlc;                    // Relocations
    unsigned short   e_cparhdr;                 // Size of header in paragraphs
    unsigned short   e_minalloc;                // Minimum extra paragraphs needed
    unsigned short   e_maxalloc;                // Maximum extra paragraphs needed
    unsigned short   e_ss;                      // Initial (relative) SS value
    unsigned short   e_sp;                      // Initial SP value
    unsigned short   e_csum;                    // Checksum
    unsigned short   e_ip;                      // Initial IP value
    unsigned short   e_cs;                      // Initial (relative) CS value
    unsigned short   e_lfarlc;                  // File address of relocation table
    unsigned short   e_ovno;                    // Overlay number
    unsigned short   e_res[4];                  // Reserved words
    unsigned short   e_oemid;                   // OEM identifier (for e_oeminfo)
    unsigned short   e_oeminfo;                 // OEM information; e_oemid specific
    unsigned short   e_res2[10];                // Reserved words
    unsigned long    e_lfanew;                  // File address of new exe header
};
```

图 1

二、NT 头

Unix V6++系统，定义 NT 头的数据结构是 ImageNTHeader。这是一个有 3 个字段的复合数据结构：

- 魔数（magic number）。Signature 字段，4 字节整数。用来标识应用程序的格式。
  - PE 格式的可执行文件，Signature 的值是 PE 标记：0x00004550（写成这样是因为 X86 芯片的整数是小端的）。展成字符串，是“PE\0\0”。
- 标准 PE 头。FileHeader 字段。这是一个 ImageFileHeader 结构，如图 2.b。记录了 PE 文件的全局属性。如该 PE 文件运行的平台、PE 文件类型(EXE / DLL)、文件中存在的逻辑段的总数等等等。ImageFileHeader 结构包含以下字段，详情请参考文献[1]。
  - Machine：指定 PE 文件运行的平台。
  - NumberOfSections：PE 文件中逻辑段（Section）的总数。这是该结构体，Unix V6++用到的唯一字段。
  - TimeDateStamp：编译器创建此文件时的时间戳。单位，秒。从 1970 年 1 月 1 日开始数秒。
  - SizeOfOptionalHeader：扩展 PE 头 ImageOptionalHeader32 结构的大小。扩展 PE 头紧贴存放在标准 PE 头之后。
  - Characteristics: 该成员为 PE 文件属性标志字段(是否可执行,是否为 DLL 等等)此值每一位都代表不同含义，请参见附录 1。
  - 其余 2 个字段关乎符号表的使用，动态链接和 debug 要用，与加载静态链接应用程序无关，此处不再赘述。
- 扩展 PE 头。OptionalHeader 字段。这是一个 ImageOptionalHeader32 结构，如图 2.c。详细描述了应用程序的内存布局（Memory Layout），也就是进程用户态虚空间的布局。ImageOptionalHeader32 结构包含以下字段，详情请参考文献[2]。
  - Magic: 魔术字,表示了 PE 文件类型，32 位程序 还是 64 位程序。
  - ImageBase：内存映像基址。这是进程虚空间中的一个基地址，ImageOptionalHeader32 结构中出现的所有其它地址是相对它的偏移量。

- AddressOfEntryPoint：程序入口地址。
  它的虚地址 = Imagebase+AddressOfEntryPoint。
- BaseOfCode：代码段起始地址。
  它的虚地址 = Imagebase+ BaseOfCode。
- BaseOfData：数据段起始地址。
  它的虚地址 = Imagebase+ BaseOfData。
- SectionAlignment：内存中（虚空间中）逻辑段的对齐粒度。Unix V6++的可执行程序是 32 位的，逻辑段 4KB 对齐。SectionAlignment = 4096。
- FileAlignment：文件中逻辑段的对齐粒度。一般是 512 字节，一个扇区的尺寸。
- SizeOfCode：代码段长度（对齐后的尺寸）。
- SizeOfInitializedData：数据段长度（带初值的数据，对齐后的尺寸）。
- SizeOfUnInitializedData：数据段长度（不带初值的数据，对齐后的尺寸）。
- SizeOfStackCommit：初始用户栈长度（应该是 4096）。
- SizeOfHeapCommit：初始堆长度。
- …… 蓝色的是 Unix V6++用到的字段。好像有瑕疵。其实，想要每个逻辑段的细节，可能应该用段表里的数据的。

```
struct ImageNTHeader
{
    unsigned long Signature;    // 0x00004550
    ImageFileHeader FileHeader;
    ImageOptionalHeader32 OptionalHeader;
};
```

```
struct ImageFileHeader
{
    unsigned short Machine;
    unsigned short NumberOfSections;
    unsigned long  TimeDateStamp;
    unsigned long  PointerToSymbolTable;
    unsigned long  NumberOfSymbols;
    unsigned short SizeOfOptionalHeader;
    unsigned short Characteristics;
};
```

程序中有几段

```
struct ImageOptionalHeader32
{
    //
    // Standard fields.
    //

    unsigned short    Magic;
    unsigned char     MajorLinkerVersion;
    unsigned char     MinorLinkerVersion;
    unsigned long     SizeOfCode;
    unsigned long     SizeOfInitializedData;
    unsigned long     SizeOfUninitializedData;
    unsigned long     AddressOfEntryPoint;
    unsigned long     BaseOfCode;
    unsigned long     BaseOfData;

    //
    // NT additional fields.
    //

    unsigned long     ImageBase;
    unsigned long     SectionAlignment;
    unsigned long     FileAlignment;
    unsigned short    MajorOperatingSystemVersion;
    unsigned short    MinorOperatingSystemVersion;
    unsigned short    MajorImageVersion;
    unsigned short    MinorImageVersion;
    unsigned short    MajorSubsystemVersion;
    unsigned short    MinorSubsystemVersion;
    unsigned long     Win32VersionValue;
    unsigned long     SizeOfImage;
    unsigned long     SizeOfHeaders;
    unsigned long     CheckSum;
    unsigned short    Subsystem;
    unsigned short    DllCharacteristics;
    unsigned long     SizeOfStackReserve;
    unsigned long     SizeOfStackCommit;
    unsigned long     SizeOfHeapReserve;
    unsigned long     SizeOfHeapCommit;
    unsigned long     LoaderFlags;
    unsigned long     NumberOfRvaAndSizes;
    ImageDataDirectory DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
};
```

描述进程用户态虚地址空间布局

程序入口地址

映像文件装入内存时，在线性地址空间中的起始地址

图 2，a、b、c

三、段表

段表中的元素是段头（SectionHeader），每个段一个。段表以一个空的 SectionHeader 结束，所以段表中有 NumberOfSections + 1 个元素。NumberOfSections 是标准 PE 头中定义的字段，表示逻辑段的数量。

```
struct ImageSectionHeader
{
    char    Name[8];
    union {
            unsigned long   PhysicalAddress;
            unsigned long   VirtualSize;
    } Misc;
    unsigned long   VirtualAddress;
    unsigned long   SizeOfRawData;
    unsigned long   PointerToRawData;
    unsigned long   PointerToRelocations;
    unsigned long   PointerToLinenumbers;
    unsigned short    NumberOfRelocations;
    unsigned short    NumberOfLinenumbers;
    unsigned long   Characteristics;
};
```

图 3

段头（ImageSectionHeader）中有 6 个字段对程序加载起重要作用：

① VirtualSize：段长。内存中对齐后的尺寸。

② VirtualAddress：内存中的起始地址。虚地址= ImageBase+ VirtualAddress。

③ SizeOfRawData：段长。文件中对齐后的尺寸。

④ PointerToRawData：文件中的偏移量。

⑤ Name 是段名，通常是以.开头的字符串。比如，代码段的名字是.text。在一个可执行文件中，不可以有同名的 2 个段。需要注意的是，段名只是一个标记，当我们要从 PE 文件中读取需要的区块时候，不能以段名作为定位的标准和依据，正确的方法是按照 IMAGE_OPTIONAL_HEADER32 结构中的数据目录字段 DataDirectory 进行定位。

⑥ Characteristics 按位指出段的属性，代码/数据/可读/可写等。UNIX V6++没有识别 Characteristics。但这个属性非常重要，常见的取值请参见附录 1，其中每个 bit 的语义请参考文献[4]。

注意 1：VirtualSize 是 4096 字节的整数倍，SizeOfRawData 是 512 字节的整数倍，一般不相等。再看 BSS 段，这个逻辑段所有变量初值是 0，无需存储，所以 BSS 段 SizeOfRawData=0，VirtualSize 是虚空间中的段长。Stack 和 heap 在文件中也无需存初值。

注意 2：SectionHeader 在段表中的排列次序与段在可执行文件中的排列次序一致。32 位静态链接的 PE 程序，段在可执行文件中的排列次序是：代码→数据→只读数据→后面的段文件中没有存储。按出现在进程中的次序：

0#段，TEXT_SECTION，代码段。

1#段，DATA_SECTION，数据段。

2#段，RDATA_SECTION，只读数据段。

3#段，BSS_SECTION，BSS 段。

Unix V6++利用这个特性，装载可执行文件各逻辑段的初值。

四、Unix V6++中负责读入、解析程序头和段表的类 PEParser

1、 PEParser 的定义

```
class PEParser
{
public:
    static const unsigned int TEXT_SECTION_IDX = 0;
    static const unsigned int DATA_SECTION_IDX = 1;
    static const unsigned int RDATA_SECTION_IDX = 2;
    static const unsigned int BSS_SECTION_IDX = 3;
    static const unsigned int IDATA_SECTION_IDX = 4;

    static const int ntHeader_size = 0xf8;
    static const int section_size = 0x28;

public:
    PEParser();
    PEParser(unsigned long peAddress);
    unsigned long Parse();
    /*
     *@comment 将Parse后的exe定位到内存中正确的位置
     *@Important 在Relocate之前需要首先调用Parse()以得到所需要的
     * exe各个section的信息，同时需要首先map好页表，否则会失败
     */
    unsigned int Relocate();
    unsigned int Relocate(Inode* p_inode, int sharedText);

    bool HeaderLoad(Inode* p_inode);

public:
    unsigned long EntryPointAddress;

    unsigned long TextAddress;
    unsigned long TextSize;

    unsigned long DataAddress;
    unsigned long DataSize;

    unsigned long StackSize;
    unsigned long HeapSize;

private:
    unsigned long peAddress;
    ImageNTHeader ntHeader;
    ImageSectionHeader* sectionHeaders;
};
```

图 4

划掉的字段已废弃不用，请自行删除。

PEParser 中的 public 类变量登记着可执行文件的解析结果：代码段和数据段的起始地址和长度，初始栈的长度和初始堆的长度，以及程序的入口地址。private 类变量是成员函数工作时需要使用的工作变量。

static const 常量硬编码了 NT 头和段头的长度，以及各逻辑段在可执行文件中的出现次序。

2、 成员函数 1：bool PEParser::HeaderLoad(Inode* p_inode)
功能：解析 PE 程序头和段表，刷新进程的内存描述符。

```
bool PEParser::HeaderLoad(Inode* p_inode)    // p_inode, 可执行文件
{
    ImageDosHeader dos_header;
    User& u = Kernel::Instance().GetUser();
    KernelPageManager& kpm = Kernel::Instance().GetKernelPageManager();

    /*读取dos header*/
    u.u_IOParam.m_Base = (unsigned char*)&dos_header;   装dos头的变量
    u.u_IOParam.m_Offset = 0;    文件中的偏移量
    u.u_IOParam.m_Count = 0x40;  读入40个字节
    p_inode->ReadI();        //文件IO不会因为多次ReadI而增加。有缓存的！从文件中读入DOS头

    /*读取nt_Header*/
    //ntHeader = (ImageNTHeader*)(kpm.AllocMemory(ntHeader_size)+0xC0000000);
    u.u_IOParam.m_Base = (unsigned char*)(&this->ntHeader);  装NT头的类变量
    u.u_IOParam.m_Offset = dos_header.e_lfanew;  文件中的偏移量
    u.u_IOParam.m_Count = ntHeader_size;         ……
    p_inode->ReadI();    从文件中读入NT头

    if ( ntHeader.Signature!=0x00004550 )
    {                                 必需PE可执行文件
        //kpm.FreeMemory(ntHeader_size, (unsigned long)ntHeader - 0xC0000000 );
        return false;
    }


    /* 原本V6++内核 : 读取Section tables至页表区。这是无奈之举，核心态用不了malloc！！
     * 希望内核用  new 和 free 函数申请动态数组。但现在的new操作符好像不对。先这么着。
     * sectionHeaders = new ImageSectionHeader;
     * */
    //sectionHeaders = (ImageSectionHeader*)(kpm.AllocMemory(section_size * ntHeader.FileHeader.NumberO
    sectionHeaders = (ImageSectionHeader*)(kpm.AllocMemory(PageManager::PAGE_SIZE * 2) + 0xC0000000);
    u.u_IOParam.m_Base = (unsigned char*)sectionHeaders;
    u.u_IOParam.m_Offset = dos_header.e_lfanew + ntHeader_size;
    u.u_IOParam.m_Count = section_size * ntHeader.FileHeader.NumberOfSections;
    p_inode->ReadI();        从文件中读入段表
```

```
    /*
     * @comment 这里hardcode gcc的逻辑
     * section 顺序为 .text->.data->.rdata->.bss
     *
     */
    this->TextAddress =
        ntHeader.OptionalHeader.BaseOfCode + ntHeader.OptionalHeader.ImageBase;
    this->TextSize =
        ntHeader.OptionalHeader.BaseOfData - ntHeader.OptionalHeader.BaseOfCode;

    this->DataAddress =
        ntHeader.OptionalHeader.BaseOfData + ntHeader.OptionalHeader.ImageBase;
    this->DataSize = this->sectionHeaders[this->IDATA_SECTION_IDX].VirtualAddress - ntHeader.OptionalHeader.BaseOfData;

    StackSize = ntHeader.OptionalHeader.SizeOfStackCommit;
    HeapSize = ntHeader.OptionalHeader.SizeOfHeapCommit;

    EntryPointAddress = ntHeader.OptionalHeader.AddressOfEntryPoint +
                        ntHeader.OptionalHeader.ImageBase;

    return true;
}
```

3、 成员函数 2：unsigned int PEParser::Relocate(Inode* p_inode, int sharedText)
功能：分配物理内存 & 系统页表构造完成后，Relocate 函数从可执行文件中依次加载代码（可执行文件的代码段），全局变量初值（可执行文件的数据段）和常数（可执行文件的只读数据段）。Unix V6++采用连续内存管理方式，Relocate 函数一次性完成整个应用程序的加载任务。

```
unsigned int PEParser::Relocate(Inode* p_inode, int sharedText) // p_inode，可执行文件。shareText，1，不用读代码段了
{
    User& u = Kernel::Instance().GetUser();
    unsigned long srcAddress, desAddress;
    unsigned cnt = 0;
    unsigned int i = 0;
    unsigned int i0 = 0;

    /* 如果可以和其它进程共享正文段，无需文件中读入正文段 */
    PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray(); // 系统的用户页表
    unsigned int textBegin = this->TextAddress >> 12 , textLength = this->TextSize >> 12;     // 计算代码段在虚空间的起始页号和页面总数
    PageTableEntry* pointer = (PageTableEntry *)pUserPageTable;

    /*如果与其它进程共享正文段，共享正文段切不可清0*/
    if(sharedText == 1)
        i = 1;      // i是段头索引 // 0#段是代码段，可以共享已有的，就不必处理它了，从1#段开始。。。
    else
    {
        i = 0;
        // 修改正文段的读写标志，为内核写代码段做准备
        for (i0 = textBegin; i0 < textBegin + textLength; i0++)
            pointer[i0].m_ReadWriter = 1;

        FlushPageDirectory();
    }

    /* 对所有页面执行清0操作，这样bss变量的初值就是0 */
    for (; i <= this->BSS_SECTION_IDX; i++ ) // 代码段，数据段，只读数据段，bss段 清0
    {
        ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
        int beginVM = sectionHeader->VirtualAddress + ntHeader.OptionalHeader.ImageBase;
        int size = ((sectionHeader->Misc.VirtualSize + PageManager::PAGE_SIZE - 1)>>12)<<12;
        int j;

//      if(sharedText == 0 || i != 0)
//      {
            for (j=0; j<size; j++)
            {
                unsigned char* b =(unsigned char*)(j + beginVM);
                *b = 0;
            }
//      }
    }
```

```
    /* 读正文段（optional）；读文件，得全局变量的初值  */
    if(sharedText == 1)
        i = 1;      // i是段头索引
    else
    // 修改正文段的读写标志，为内核写代码段做准备
        i = 0;

    for ( ; i < this->BSS_SECTION_IDX; i++ ) // 读可执行文件的代码段，数据段 和 只读数据段
    {
        ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
        srcAddress = sectionHeader->PointerToRawData;
        desAddress =
            this->ntHeader.OptionalHeader.ImageBase + sectionHeader->VirtualAddress;

        u.u_IOParam.m_Base = (unsigned char*)desAddress;
        u.u_IOParam.m_Offset = srcAddress;
        u.u_IOParam.m_Count = sectionHeader->Misc.VirtualSize; // 好像错了，应该是 SizeOfRawData

        p_inode->ReadI();

        cnt += sectionHeader->Misc.VirtualSize;
    }

    if(sharedText == 0)
    {   //将正文段页面改回只读
        for (i0 = 0; i0 < textLength; i0++)
            pointer[i0].m_ReadWriter = 0;

        FlushPageDirectory();
    }

    KernelPageManager& kpm = Kernel::Instance().GetKernelPageManager();    // 释放从页表区借来的，暂时用来装段表的那两个物理页框
    kpm.FreeMemory(PageManager::PAGE_SIZE * 2, (unsigned long)this->sectionHeaders - 0xC0000000 );
//  kpm.FreeMemory(section_size * ntHeader.FileHeader.NumberOfSections, (unsigned long)this->sectionHeaders - 0xC0000000 );
//  delete this->sectionHeaders;
    return  cnt;    // 总共读了多少字节呢？
}
```

Part 2、Exec 系统调用

PPT、课本已阐释详尽，不再赘述。

附录 1：PE 文件的属性字段。标准 PE 头中的 Characteristics 字段

数据位	常量符号	常量值	为1时的含义
0	IMAGE_FILE_RELOCS_STRIPPED	0x0001	文件中不存在重定位信息
1	IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	文件是可执行的
2	IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	不存在行信息
3	IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	不存在符号信息
4	IMAGE_FILE_AGGRESIVE_WS_TRIM	0x0010	调整工作集
5	IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	应用程序可处理大于2GB的地址
6			此标志保留
7	IMAGE_FILE_BYTES_REVERSED_LO	0x0080	小尾方式
8	IMAGE_FILE_32BIT_MACHINE	0x0100	只在32位平台上运行
9	IMAGE_FILE_DEBUG_STRIPPED	0x0200	不包含调试信息
10	IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	0x0400	不能从可移动盘运行
11	IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	不能从网络运行
12	IMAGE_FILE_SYSTEM	0x1000	系统文件(如驱动程序),不能直接运行
13	IMAGE_FILE_DLL	0x2000	这是一个DLL文件
14	IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	文件不能在多处理器的计算机上运行
15	IMAGE_FILE_BYTES_REVERSED_HI	0x8000	大尾方式

附录 2：PE 文件中逻辑段的属性。ImageSectionHeader 中 Characteristics 字段

数值	含义
IMAGE_SCN_CNT_CODE 0x00000020	The section contains executable code. 包含代码，常与 0x10000000 一起设置。
IMAGE_SCN_CNT_INITIALIZED_DATA 0x00000040	The section contains initialized data. 该区块包含以初始化的数据。
IMAGE_SCN_CNT_UNINITIALIZED_DATA 0x00000080	The section contains uninitialized data. 该区块包含未初始化的数据。
IMAGE_SCN_MEM_DISCARDABLE 0x02000000	The section can be discarded as needed. 该区块可被丢弃，因为当它一旦被装入后，进程就不在需要它了，典型的如重定位区块。
IMAGE_SCN_MEM_SHARED 0x10000000	The section can be shared in memory. 该区块为共享区块。
IMAGE_SCN_MEM_EXECUTE 0x20000000	The section can be executed as code. 该区块可以执行。通常当 0x00000020 被设置 时候，该标志也被设置。
IMAGE_SCN_MEM_READ 0x40000000	The section can be read. 该区块可读，可执行文件中的区块总是设置该标志。
IMAGE_SCN_MEM_WRITE 0x80000000	The section can be written to. 该区块可写。

把这些 bit 组合起来就可以得到常用的段的属性。比如，不可写、有初始化数据的段是只读数据段。可以观察系统中的可执行文件，读出来每个段的 characteristics，加深理解。

参考文献：

1、3.PE 文件之标准 PE 头(IMAGE_FILE_HEADER)_
https://blog.csdn.net/m0_46125480/article/details/120954398
2、4.PE 文件之扩展 PE 头(IMAGE_OPTIONAL_HEADER)
https://blog.csdn.net/m0_46125480/article/details/120975889
3、
https://blog.csdn.net/chenlycly/article/details/53378196?utm_medium=distribute.pc_feed_404.none-task-blog-2~default~BlogCommendFromBaidu~Rate-4-53378196-blog-null.pc_404_mixedpudn&depth_1-utm_source=distribute.pc_feed_404.none-task-blog-2~default~BlogCommendFromBaidu~Rate-4-53378196-blog-null.pc_404_mixedpud
4、
https://blog.csdn.net/weixin_34075551/article/details/94641142