2021/03/03_PE_第12课_资源表

笔记本: PE
创建时间: 2021/3/3 星期三 10:01
作者: ileemi

在可执行文件中静态使用TLS为什么一定会申请成功?
每次开一个线程都会申请一定量的内存空间共TLS数组使用。每一个线程对应一个对应的TLS数组。

操作系统一般支持的最大线程总数为65535。

资源表

不同操作系统对资源的设计是不一样的。资源的设计可以存放在可执行文件中也可以存放在单独的目录中。在 Windows上可执行文件的资源是存放在其可执行文件中的。

资源数据保存在 ".res" 文件中（自定义的资源（自定义的资源需要定义一个自定义的资源类型）的数据也保存在该文件中），资源数据中的字符串都是以Unicode进行存储的。

图片、对话框二进制数据中可以存放代码（精心设计）。

编译器在编译一个程序的时候会对其资源进行分类

> 选项头中的数据目录的**第三项**是资源表:
> **IMAGE_DIRECTORY_ENTRY_RESOURCE**

资源是分类（有对应的宏）的:

- RT_CURSOR
- RT_BITMAP
- ...

```c
/*
 * Predefined Resource Types
 */
#define RT_CURSOR           MAKEINTRESOURCE(1)
#define RT_BITMAP           MAKEINTRESOURCE(2)
#define RT_ICON             MAKEINTRESOURCE(3)
#define RT_MENU             MAKEINTRESOURCE(4)
#define RT_DIALOG           MAKEINTRESOURCE(5)
#define RT_STRING           MAKEINTRESOURCE(6)
#define RT_FONTDIR          MAKEINTRESOURCE(7)
#define RT_FONT             MAKEINTRESOURCE(8)
#define RT_ACCELERATOR      MAKEINTRESOURCE(9)
#define RT_RCDATA           MAKEINTRESOURCE(10)
#define RT_MESSAGETABLE     MAKEINTRESOURCE(11)

#define DIFFERENCE      11
#define RT_GROUP_CURSOR MAKEINTRESOURCE((ULONG_PTR)(RT_CURSOR) + DIFFERENCE)
#define RT_GROUP_ICON   MAKEINTRESOURCE((ULONG_PTR)(RT_ICON) + DIFFERENCE)
#define RT_VERSION      MAKEINTRESOURCE(16)
#define RT_DLGINCLUDE   MAKEINTRESOURCE(17)
#if(WINVER >= 0x0400)
#define RT_PLUGPLAY     MAKEINTRESOURCE(19)
#define RT_VXD          MAKEINTRESOURCE(20)
#define RT_ANICURSOR    MAKEINTRESOURCE(21)
#define RT_ANIICON      MAKEINTRESOURCE(22)
#endif /* WINVER >= 0x0400 */
#define RT_HTML         MAKEINTRESOURCE(23)
#ifdef RC_INVOKED
#define RT_MANIFEST     24
```

操作系统对资源结构的设计是按照 "**文件目录**" 的方式进行设计的。资源结构进行分类存储的时候（可以使用ID号分类存储也可以使用字符串分类存储）**为了节省内存空间一般采用ID号（16进制的数值）进行分类存储。**

资源自定义存储，使用ID分类存储，ID号需要在25项（0~24）之后，前25项的ID号已被操作系统定义并使用。

资源表结构体

> 资源表目录最少需要3层（资源根目录、资源类别、对应的资源项）。在设计上没有说每类具体的层数，最少三层（目录中存在目录），最多没有上限（支持嵌套）。

```
IMAGE_RESOURCE_DIRECTORY
IMAGE_RESOURCE_DIRECTORY_ENTRY[]
{
    IMAGE_RESOURCE_DIRECTORY
    IMAGE_RESOURCE_DIRECTORY_ENTRY[]
    {
        IMAGE_RESOURCE_DIRECTORY
        IMAGE_RESOURCE_DIRECTORY_ENTRY[]
        {
            IMAGE_RESOURCE_DATA_ENTRY Data
            ...
        }
    }
}
```

> IMAGE_RESOURCE_DIRECTORY_ENTRY[**NumberOfNamedEntries + NumberOfIdEntries**]

IMAGE_RESOURCE_DIRECTORY

结构体定义如下（总大小为16字节）：

```
/*
资源目录由两个计数组成，后跟一个可变长度目录项数组。
第一个计数是在具有与每个条目关联的实际名称的数组的开头。
条目按升序排列，不区分大小写。
第二个  count是紧跟在命名条目之后的条目数。第二个计数标识具有16位整数的条目数
ID作为他们的名字。这些条目也按升序排序。
这种结构允许通过名称或数字进行快速查找，但对于任何给定的资源条目只支持一种形式的查找，
而不支持两种形式。这与.RC文件和.RES文件的语法一致。
*/
typedef  struct  _IMAGE_RESOURCE_DIRECTORY  {
    DWORD  Characteristics; // 属性，参考
    DWORD  TimeDateStamp; // 时间日期戳，参考
    WORD  MajorVersion; // 主版本号，参考
    WORD  MinorVersion; // 次版本号，参考
    WORD  NumberOfNamedEntries; // 以名称（字符串）命名的入口数量，重要
    WORD  NumberOfIdEntries; // 以ID（整型数字）命名的入口数量，重要
//    IMAGE_RESOURCE_DIRECTORY_ENTRY  DirectoryEntries[];
}  IMAGE_RESOURCE_DIRECTORY,  *PIMAGE_RESOURCE_DIRECTORY;
```

> 紧随其后的是 **IMAGE_RESOURCE_DIRECTORY_ENTRY** 结构。
> NumberOfNamedEntries + NumberOfIdEntries = DirectoryEntries目录项的总和

IMAGE_RESOURCE_DIRECTORY_ENTRY

结构体定义如下：

```
//  判断表项的最高位，1--目录，0--不是目录
#define  IMAGE_RESOURCE_NAME_IS_STRING  0x80000000
#define  IMAGE_RESOURCE_DATA_IS_DIRECTORY  0x80000000

/*
每个目录包含条目的32位名称和偏移量，相对于关联数据的资源目录的开头使用此目录项。
如果条目的名称是实际文本字符串而不是整数Id，然后是名称字段的高位设置为1，
低阶31位是相对于字符串的资源目录的开头，类型为图像\资源\目录\字符串。
否则高位清零低位16位是标识此资源目录的整数Id进入。
```

```
如果目录项是另一个资源目录（即子目录），则偏移字段的高位设置为指示此情
况。
否则高位清零，偏移量字段指向资源数据项
*/

typedef  struct  _IMAGE_RESOURCE_DIRECTORY_ENTRY  {
    union  {
        struct  {
            DWORD  NameOffset:31;
            DWORD  NameIsString:1;
        }  DUMMYSTRUCTNAME;
        DWORD  Name;
        WORD  Id;
    }  DUMMYUNIONNAME;
    union  {
        DWORD  OffsetToData;
        struct  {
            DWORD  OffsetToDirectory:31;
            DWORD  DataIsDirectory:1;
        }  DUMMYSTRUCTNAME2;
    }  DUMMYUNIONNAME2;
}  IMAGE_RESOURCE_DIRECTORY_ENTRY,  *PIMAGE_RESOURCE_DIRECTORY_ENTRY;

// 可以简单理解为：
typedef  struct  _IMAGE_RESOURCE_DIRECTORY_ENTRY  {
    //  最高位为1，指向名称(_IMAGE_RESOURCE_DIR_STRING_U)
    //  最高位为0，指向ID
    DWORD  NameOffset;
    //  最高位为1，指向下一层目录(IMAGE_RESOURCE_DIRECTORY)
    //  最高位为0，指向数据(IMAGE_RESOURCE_DATA_ENTRY)
    DWORD  OffsetToData;
}  IMAGE_RESOURCE_DIRECTORY_ENTRY,  *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

> 偏移均为节内偏移，也就是 OffsetToData == IMAGE_RESOURCE_DIRECTORY(资源目录的首地址) + RVA

IMAGE_RESOURCE_DIR_STRING_U

结构体定如下：

```
/*
对于具有实际字符串名称的资源目录项，名称目录条目的字段指向以下类型的对
象。
所有这些字符串对象都存储在最后一个资源之后目录项和第一个资源数据对象之
前。
```

```
这将最小化这些可变长度对象对固定对象对齐的影响调整目录项对象的大小。
*/

typedef  struct  _IMAGE_RESOURCE_DIR_STRING_U  {
    WORD  Length;  //  字符串长度
    WCHAR  NameString[  1  ];  //  字符串数组
}  IMAGE_RESOURCE_DIR_STRING_U,  *PIMAGE_RESOURCE_DIR_STRING_U;
```

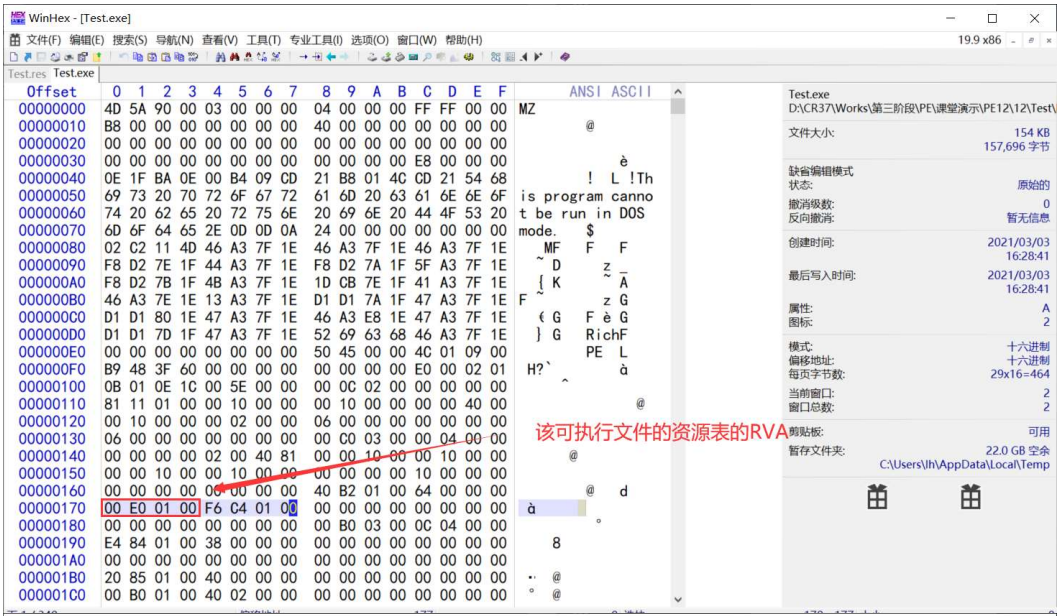
IMAGE_RESOURCE_DATA_ENTRY

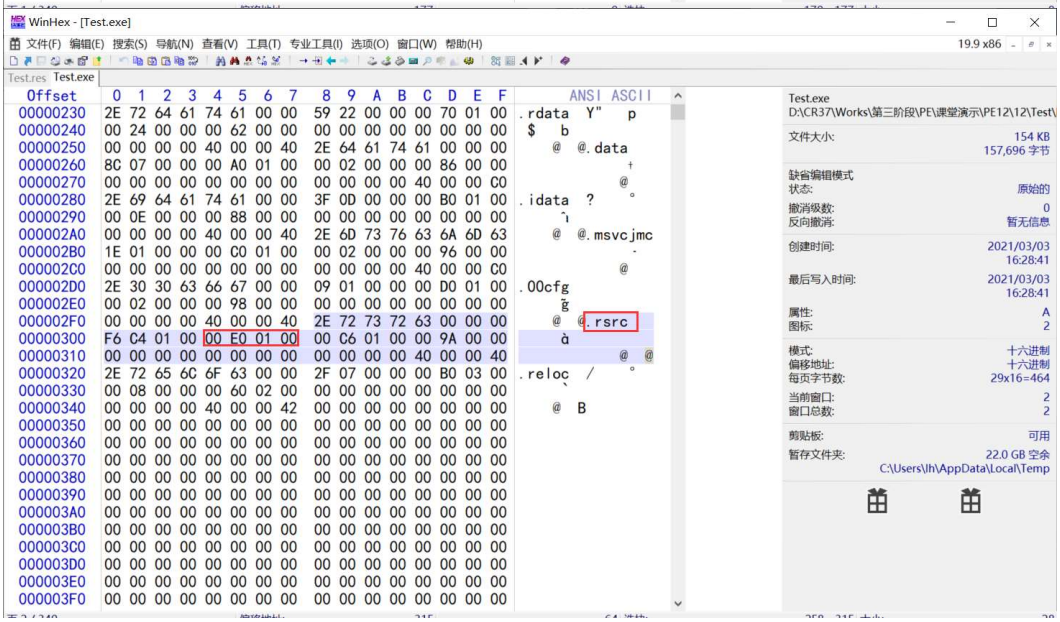
结构体定如下：

```
/*
每个资源数据项描述资源目录中的叶节点树。它包含相对于资源开头的偏移量资源数据的目录，
给出数字的大小字段在该偏移量处的数据字节数，当对资源数据中的码点值进行解码。
通常用于新的应用程序的代码页将是unicode代码页。
*/
//@[comment("MVI_tracked")]
typedef  struct  _IMAGE_RESOURCE_DATA_ENTRY  {
    DWORD  OffsetToData;  //  资源数据的RVA  Image + RVA
    DWORD  Size;  //  资源数据的长度
    DWORD  CodePage;  //  代码页(一般指字符集)
    DWORD  Reserved;  //  保留字段
}  IMAGE_RESOURCE_DATA_ENTRY,  *PIMAGE_RESOURCE_DATA_ENTRY;
```

分析示例

> 资源表有对应的节区（.rsrc）

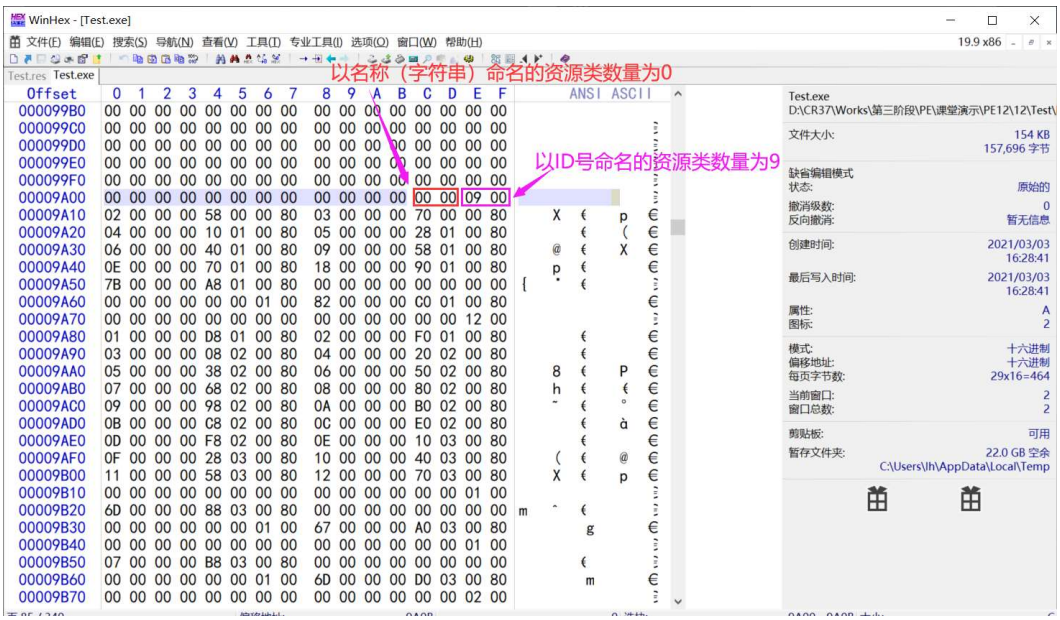




当前程序的资源表的RVA：0x1E000（位于 ".rsrc" 节中）

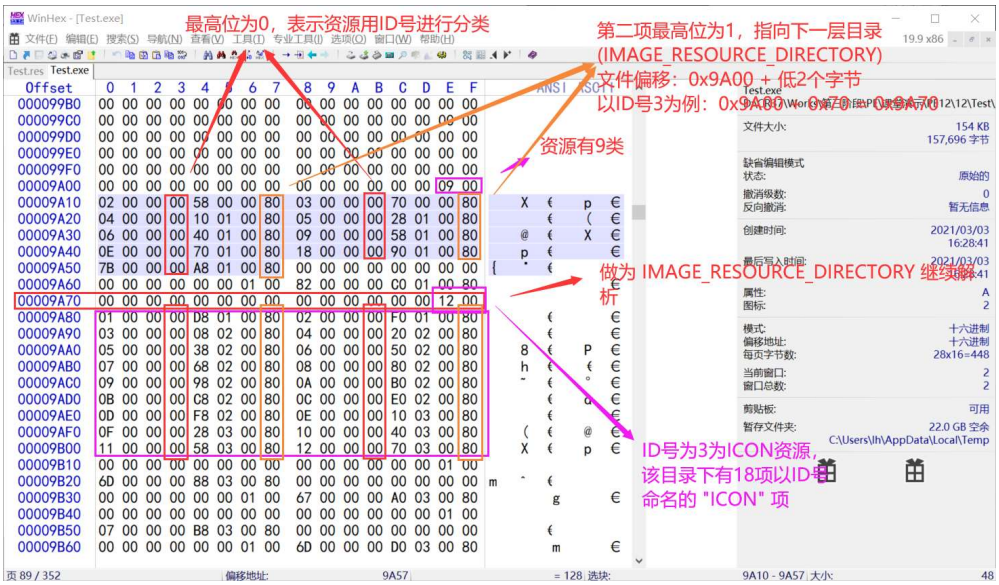
FA：0x9A00

确定资源命名的数量，该文件中以名称命名的数量为0，以ID号命名的数量为9：

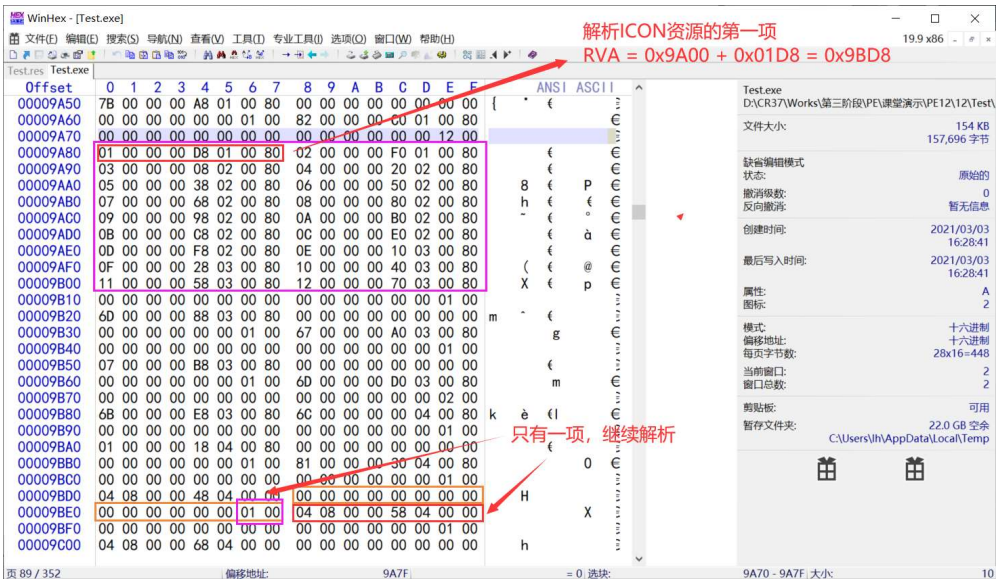


解析ID号为3的资源：

- 解析第一层：

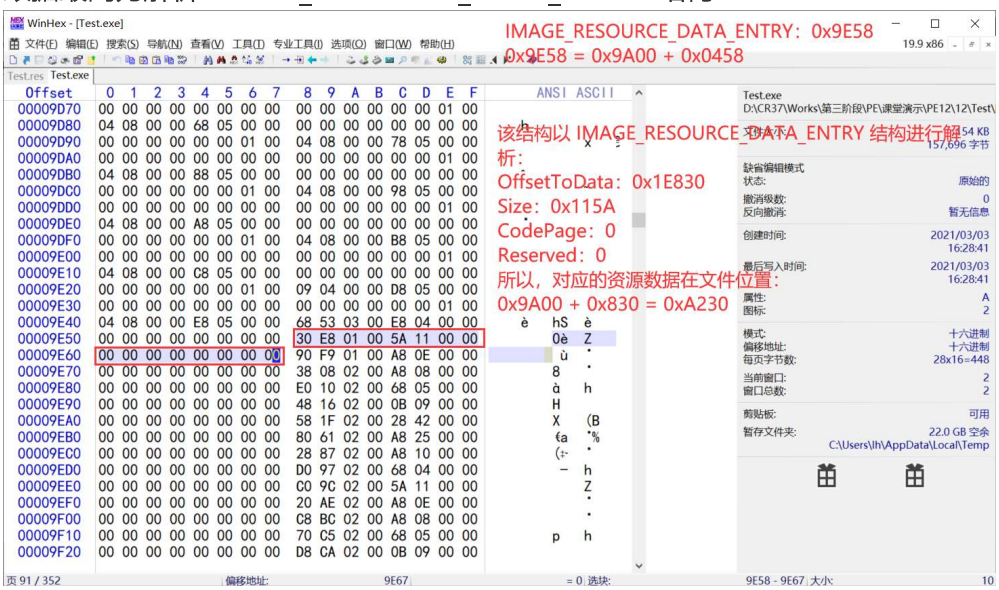


- 解析第二层：



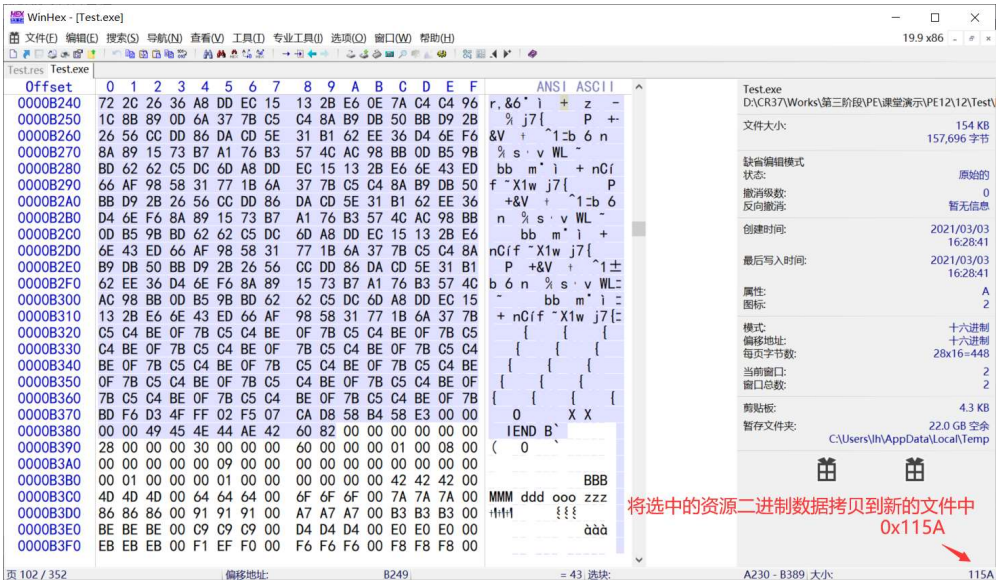
- 解析第三层：

- 跟据最高为解析 IMAGE_RESOURCE_DATA_ENTRY 结构：

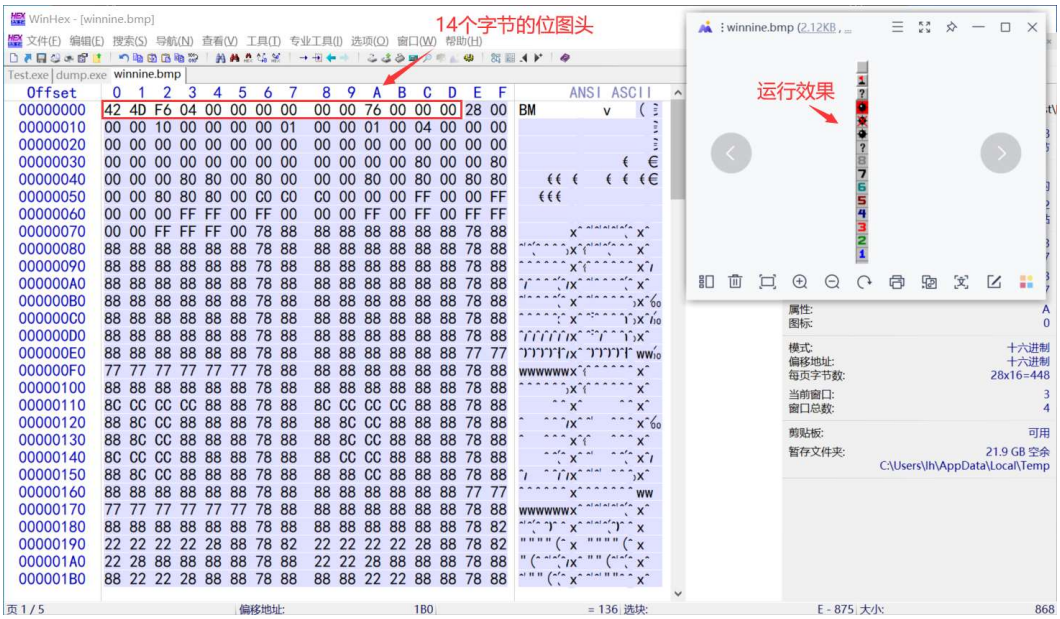


- 在文件中找到资源数据的位置：访问0xA230

- 按照该资源数据的大小（0x115A）将其二进制代码拷贝到一个新的文件中并尝试打开：



> 在可执行文件中（Windows），位图的二进制数据并没有保存位图的结构体头部（标志：14个字节），只保存了图片的所有像素点、颜色等数据。



补课

> 回顾PE课程

- Dos头：兼容老版本的可执行文件（16位的可执行程序）。
- NT头（分32位、64位），包含文件头、选项头（区分32位、64位）。
- 节：节的数量由文件头决定，节的起始位置 = 选项头 + 选项头的大小。加节扩展文件内存后，一般会影响选项头的 "SizeOfImage" 成员。节的数量也需要++，镜像大小也要进行判断（根据对齐值判断）。扩展最后一个节的内存大小就不会影响到 "ImageBase" 成员。最后一个节后面没有空间添加新的节，就需要修改 "SizeOfHeaders" 的大小，同时所有已存在的节的 "FA" 也要进行修改，"RVA" 不变（注意文件对齐值和内存对齐值是否一样，一样就需要修改）。
- 数据目录（重要的表）：
  导出表：dll或者可执行文件（不常见），库名称、函数名称表（项数和函数序号表一致）、函数序号表、函数地址表。**函数名称导出**，查询.函数名称表 -->函数序号表 --> 函数地址表。**序号导出**，查询 序号表 - 起始序号 --> 函数地址表。
  导入表：表结构：库名称、函数名称表（可以为NULL）、 IAT表（有效不可以为NULL）
  资源表：表结构：目录、目录项、数据项
  重定位表：dll或者随机基址的可执行文件

> 选项头是32位结构还是64位结构在文件头中可以由文件头对应的结构体成员去决定（运行平台+**文件类型**）。
> 在选项头中就需要**Magic**成员来判断该可执行文件状态（16位、32位）。