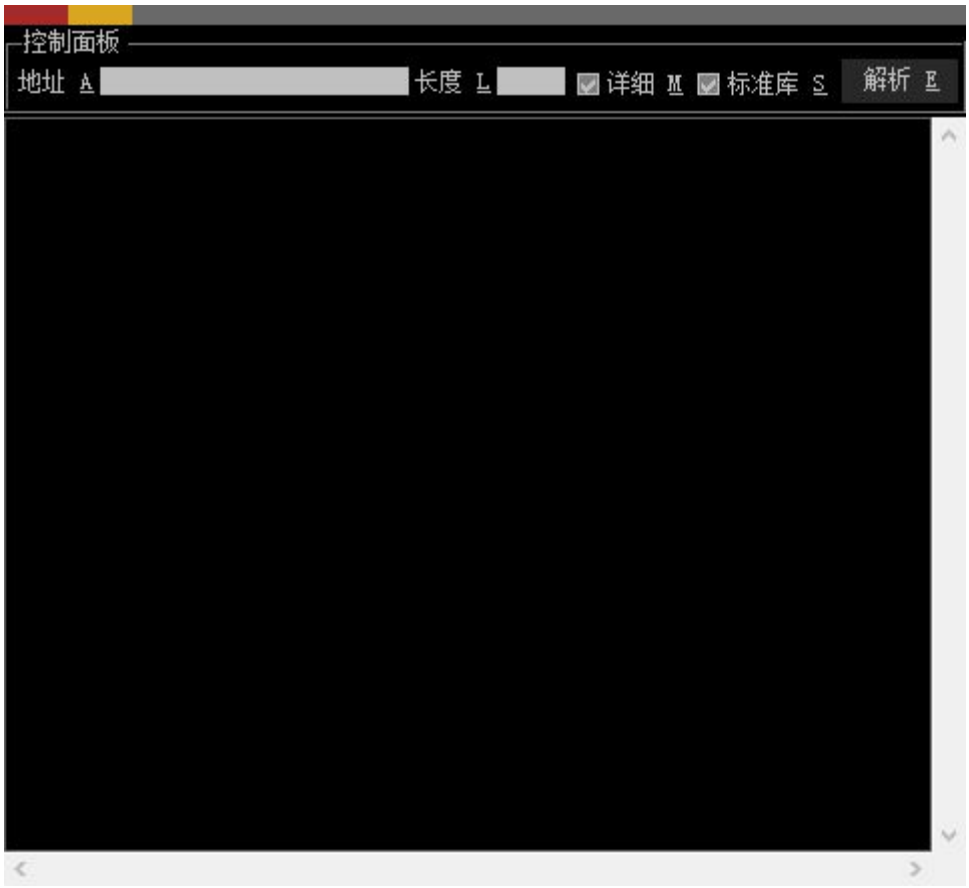
RTTI Extractor 使用说明

- RTTI Extractor 是一个 DLL 动态库，需要注入进程使用。（怎么注入？自己想办法）
- RTTI Extractor 有 x86 和 x64 两个版本
- RTTI Extractor 需要 Microsoft .NET Framework 4.0及以上支持，需要 VC++2013(x86/x64)运行时库支持。
- 以下是 RTTI Extractor 的启动界面：（x86/x64版本界面一致）



- 窗口上部控件分别为：【关闭 C】、【最小化 N】、【窗口移动】。

- 窗口不能改变大小。占据窗口大部分区域的文本控件是【信息输出控件】，双击将清除所有数据
- 

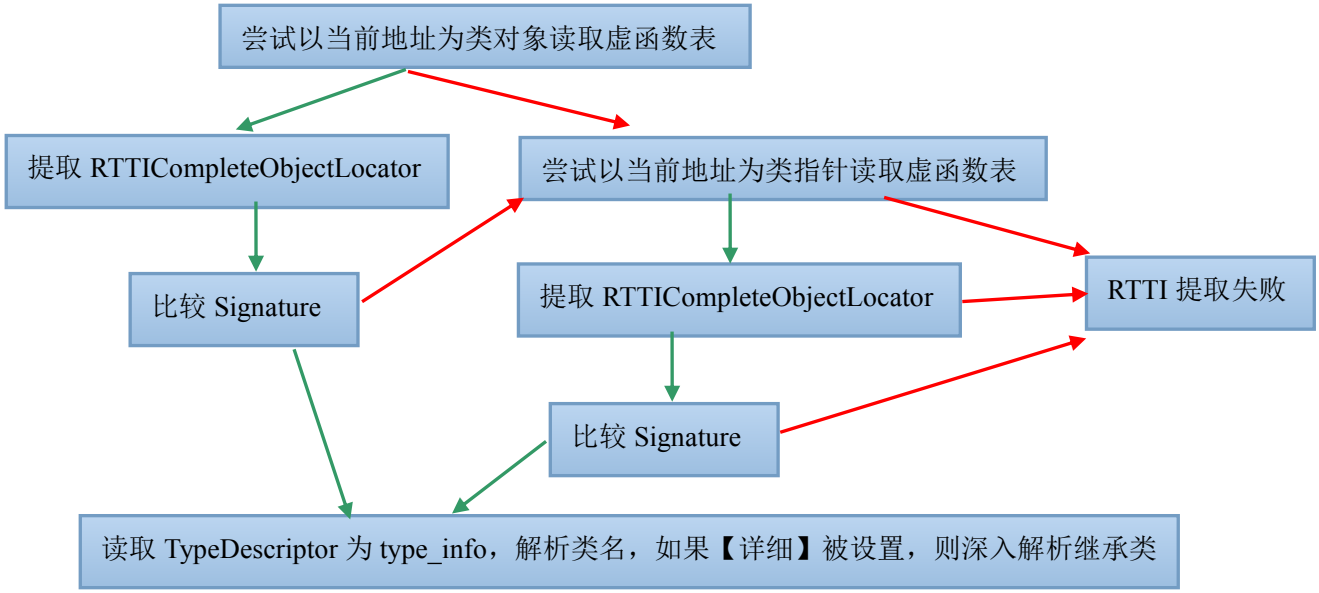
【地址 A】接受需要解析的内存起始位置（十六进制）

- 

【长度 L】为解析的数据长度（十六进制），限制为 0~FFFF，缺省为 0x200
- 【详细 M】被选中时，将解析继承的类，以及指向标准库的指针
- 【标准库 S】被选中时，解析类失败后，将尝试解析为标准库对象
- 单击【解析 E】将开始解析过程
- 解析类需要类有虚表，且存在 RTTI 信息，否则无法识别。RTTI 信息的识别基础如下（针对 x64 有额外处理）：

```
struct TypeDescriptor
  {
  DWORD ptrToVTable;
  DWORD spare;
  char name[8];
  };
struct PMD
  {
  int mdisp;  //member displacement
  int pdisp;  //vbtable displacement
  int vdisp;  //displacement inside vbtable
  };
struct RTTIBaseClassDescriptor
  {
  struct TypeDescriptor* pTypeDescriptor; //type descriptor of the class
  DWORD numContainedBases; //number of nested classes following in the Base Class Array
  struct PMD where;       //pointer-to-member displacement info
  DWORD attributes;       //flags, usually 0
  };
struct RTTIClassHierarchyDescriptor
  {
  DWORD signature;     //always zero?
  DWORD attributes;     //bit 0 set = multiple inheritance, bit 1 set = virtual
inheritance
  DWORD numBaseClasses; //number of classes in pBaseClassArray
  struct RTTIBaseClassArray* pBaseClassArray;
  };
struct RTTICompleteObjectLocator
  {
  DWORD signature; //always zero ?
  DWORD offset;    //offset of this vtable in the complete class
  DWORD cdOffset;  //constructor displacement offset
  struct TypeDescriptor* pTypeDescriptor; //TypeDescriptor of the complete class
  struct RTTIClassHierarchyDescriptor* pClassDescriptor; //describes inheritance
hierarchy
  };
```

- RTTI 处理细节如下：



- 如果类名起始为“class”，输出信息为“●”，如类“class A”，将输出为“●A”
- 如果类名起始为“struct”，输出信息为“○”，如结构体“struct A”，将输出为“○A”
- 未知类名输出信息为“◆”，如类“class D : public A , public B, public C”，将输出为“●D： ●A， ●B， ●C”
- 输出信息“↓”表示该地址为类对象起始，而“→”表示为该地址为类对象指针
- 当处理 RTTI 失败后，且指定解析【标准库】时，将开始尝试解析为如下对象类型：

  wstring、string、map、vector（这是解析尝试的先后顺序）
- wstring 输出为“〓”、string 输出为“≈”，解析结构如下：

```
    _BUF_SIZE = 16 / sizeof (value_type) < 1 ? 1
          : 16 / sizeof (value_type)};
    union _Bxty
        {   // storage for small buffer or pointer to larger one
        value_type _Buf[_BUF_SIZE];
        pointer _Ptr;
        char _Alias[_BUF_SIZE]; // to permit aliasing
        } _Bx;

    size_type _Mysize;  // current length of string
    size_type _Myres;   // current storage reserved for string
```

  对 wstring、string 的识别条件如下：

```
1. 0<=size()< capacity()<=0x00800000
2. _tcslen(c_str()) == size()
3. capacity()合法
```

- vector 输出为“☆”，解析结构如下：

```
struct f_vector
  {
    const unsigned char* _Myfirst;
    const unsigned char* _Mylast;
    const unsigned char* _Myend;
  };
```

对 vector 的识别条件如下：

```
1. _Myfirst、_Mylast、_Myend 为有效指针
2. _Myfirst<=_Mylast<=_Myend
3. size()<=0x00800000
```

- map 输出为“★”，解析结构如下：

```
struct f_map_node
  {
    const f_map_node* _Left;
    const f_map_node* _Parent;
    const f_map_node* _Right;
  };
struct f_map
  {
    const f_map_node* _Myhead;
    size_t            _Mysize;
  };
```

对 map 的识别条件如下：

```
1. _Myhead 为有效指针
2. 0<=_Mysize<=0x00800000
3. _Left、_Parent、_Right 为有效指针，以及其下一层也必须为有效指针
```

- 输出信息“↓”表示该地址为类对象起始，而“→”表示为该地址为类对象指针
- 注意：因为只要符合结构和条件就算解析成功，所以标准库的解析不一定会正确。
- 以下为解析示例