首先因为这是一个比较大的项目，而且开发人员可能有四个，所以我觉得需要一个文档记录自己开发的想法，我自己也没有写过类似的文档，只能试着写，然后也希望每一位开发者都能够多记载一些文档，描述自己的开发思路和接口使用方法，此外为了程序的健壮性，请在每一步都思考需不需要进行错误检查，为一个类添加一个变量后是否要考虑其资源释放，要不要为其发送相应的变动信号等等。

整个程序的一些设想：

为了方便分工，整个程序参照MVC模式设计，主要想法就是后台负责解析pe文件内容，将其保存在一个buffer数组中，由各个指针指向相应的数据结构，UI负责将这些指针指向的内容映射到合适的UI控件中，在UI控件中的改动能保存到数据结构中，此外还需要根据用户的一些操作对相应的控件进行设置，比如用户没有改动数据结构的内容，那保存按钮的使能状态就要是false，如果改动了那使能状态就要变成true，所以需要大量用到信号与槽机制进行控制与通信。

然后还有关于错误状态的提醒，由于后台和UI是完全分开的，仅靠BOOL指示操作是否完成可能会造成混乱，比如程序有很多地方return false，当出错时到底是哪儿返回，比较难定位，一开始我想采用GetLastError形式，但是要解决错误处理函数可重入性与线程安全的问题，所以所有函数的返回参数采用QString，””表示正确，否则返回错误原因，UI的开发者可以直接使用QMessageBox::information进行提示。

一些开发规范：

为了一致，请采用如下规范：

①：对于private成员，采用java setter和getter起名规则，即get+成员名称（成员名称首字母大写）

②：错误信息统一返回QString，不要使用BOOL或INT，除非迫不得已。

**GlobalStore类：**

**globalStore类**主要是存储一些横跨全局的信息，比如上面的peFile类，采用单例模式设计，使用时用getInstanceP获取指针，**需要注意的是，凡是包含在该类中的类，不应该也不能使用其构造函数构造其类**。

成员:

Options\* options;

peFile\* pefile;

接口:

**getInstanceP()**

获取实例指针，此类不应使用构造函数而应该使用该方法获取实例，且永远只存在一个实例。

**Options类：**

**Options类**主要用于存储一些程序的设置（比如后续可能允许用户设置字体等等，但是由于时间缘故大概率可能没有），为了方便，所有成员为public

目前主要存储程序加载目标文件的路径，成员名称为：pePath

**peFile：**

**peFile类**是后台最重要的类之一，该类主要存储目标文件的二进制内容并将其自动解析为pe结构并发送相应的信号。

成员:

peBuffer（存储pe文件的二进制内容）、

peFileSize（pe文件的大小）、

modifyFlag（标识程序是否对pe文件进行修改，这个标识每次变动都会发出信号（其它开发者也请遵守），具体参见注释，可以做什么，比如，你可以将这个信息绑定到UI保存按钮的setEnable函数上自动实现按钮的使能控制，也可以在程序发出exit信号/加载其它文件时，检查这个标识看用户是否要退出/加载其它文件，如果标识是true，那你就要警告用户，说你还没有保存，然后问它是否要保存，是的话调用writebuffer函数写回）。

peFlag32 表示是否为32位的文件

一些pe结构相关的数组（均为指针且直接指向peBuffer）

**接口:**

**setPeBuffer(**QString & path**)** 传入路径，自动读入文件，解析文件结构，填充peFile其它成员，会自动修改modifyFlag标志为false

**writeBuffer(**QString & path**)** 按照路径写回文件，会自动修改modifyFlag标志为false

**updatePointer()** 解析peBuffer中的内容，自动设置peFile中的相关成员，将会发送updateFile信号，可以将此信号与UI更新的函数进行绑定，实现当peBuffer重新加载时，UI相应控件也会自动更新。(目前此函数还未写完，还有资源节、异常节没有解析)

**clear()** 此函数用于清理peFile相关的信息，如当程序加载新文件时，将需要调用此函数加载前面已经加载的文件的相关信息，并会发送peFileClear信号，可以将此信号与UI的相关控件槽函数进行绑定实现自动清理控件内容。

**getPe32Flag()类型函数，获取标志位**

**getNtHeader64()类型函数，获取Nt头，且为64位。**

**信号:**

**modifyFlagChanged\_signal(**bool modifyFlag**)** 当modifyFlag变量发生变化时需要发送该信号，该信号表明程序对加载的pe文件内容进行了修改或者保存，flag为false表示已保存或重新加载，为true表明有程序对pe文件的内容进行了保存。

**updateFile()** 该信号表明peBuffer的内容可能发生了变换，并且peFile中的pe文件结构一定发生了变化，可以将此信号绑定在UI的更新函数上更新UI。

**peFileClear()** 此信号表明peFile相关信息已经被清理，需要重新加载

**Util：**

**Util类**主要存储一些工具函数。

**成员：**

**接口：**

DWORD **getFileOffsetByRVA**(peFile \* pe, DWORD RVA);

根据RVA获取该地址相对于文件首部的偏移

int **getSectionIndexByName**(peFile \* pe, const QString & name);

根据名称获取节在节表中的索引

QList<opClass32>\* **Disassembly32**(BYTE opCode[], long size,unsigned int VirtualAddr);

QList<opClass64>\* **Disassembly64**(BYTE opCode[], long size,unsigned long long VirtualAddr);

opClass32和opClass64定义如下：

typedef struct **OPCLASS32**{

unsigned int Addr; //地址

QString opCode; //汇编机器码

QString opInstr; //汇编语句

}opClass32, \*pOpClass32;

typedef struct **OPCLASS64**{

unsigned long long Addr; //va地址，从传入的virtualAddr开始

QString opCode; //汇编机器码

QString opInstr; //汇编语句

}opClass64, \*pOpClass64;

函数声明分别传入：代码节数组、代码节大小、开始va，调用示例：

1. **int** i = g->pefile->getSectionIndexByName(".text");
2. **if**(i == -1){
3. qDebug() << "获取索引失败!";
4. **return** 0;
5. }
6. //测试反汇编功能
7. QList<opClass64> \*tmp =
8. Util::Disassembly64(g->pefile->getSectionArray()[i], g->pefile->getSectionHeaderArray()[i].Misc.VirtualSize,
9. g->pefile->getSectionHeaderArray()[i].VirtualAddress+g->pefile->getNtHeader64()->OptionalHeader.ImageBase);
11. //使用
12. //使用完毕
13. **delete** tmp;

注意要自己释放返回的QList指针**，delete即可（不要用free）**

static QString **PeAttach**(QString & path);

//传入希望附加的文件路径

返回错误信息。

**ParsePe类：**

该类主要实现了一些解释数字含义或者把字符串变成数字的函数。

**属性：**

QHash<DWORD,QString> machineDSTable

存储了机器类型码和机器的哈希表，可以通过

machineDSTable.values()获取类型码表

machineDSTable.keys()获取机器名称表

**方法：**

DWORD **parseStringToTimeDateStamp**(QString & time)：

将2016-09-28 23:59:02格式的时间解析为DWORD，用于用户修改文件中以DWORD形式出现的时间(由用户选择时间，然后开发者调用这个再写回)。

QString **parseTimeDateStampToString**(DWORD time)：

将DWORD形式的数字解释为时间，格式和上个函数一致。

QString **parseMachine**(DWORD machine)：

将FileHeader.Machine解释为字符串，如0x0200表示为IMAGE\_FILE\_MACHINE\_IA64

DWORD **parseQStringToMachine**(QString & machine)

将String解析为数字，如“IMAGE\_FILE\_MACHINE\_IA64”返回IMAGE\_FILE\_MACHINE\_IA64

关于UI展示的一些记录：

参考peview展示ui的方式，肯定采用了table维护数据，在Qt开发中，一般采用model/view维护表格，为了向model插入数据，需要生成非常多的**QStandardItem**对象，为了效率，需要建立多个这样的对象，这样会导致非常非常多的内存被浪费，因为有多个不同的ui，所以有多个不同的model，内存占用高达4-5GB,所以需要修改数据的维护方式，不能生成QStandardItem，尽量从存储数据的buffer里直接读取数据，还能解决数据一致性的问题。所以重写了QAbstractTableModel类。