接着上一篇文章，我们来聊关于整个辅助框架的代码实现

Stu

按照常规的思路，我们来构想一下接下来的开发过程。首先我们需要找到人物的数据，然后编写代码的时候设计一个人物数据的结构体；接着周围遍历的数据，又要设计一个周围遍历的结构体；后续用到每一个新的数据，都要设计一个结构体来存放。

等写了两三个游戏以后就会发现，很多代码都在重复写。那么这就引出了一个数据管理的问题，怎么样用最高效的方式去管理这些数据。

其实很多对象有一部分属性都是通用的，比如说名字，等级和ID；这样我们就可以设计一个通用的对象结构体，这个对象结构体存储所有的游戏数据对象，然后用一个type字段来对对象进行区分。

对象结构如下：

```
struct _stuObj
{
    int m_StuType;          //0 人物
    //--------------------------------公用-------------------------------------
----------
    wstring m_Name;             //名字
    DWORD m_Obj;                //对象
    DWORD m_ID;                 //ID

    //输出调试信息
    void OutputDebugInfo();
}
```

1. m_StuType表示对象的类型，用这个字段来区分不同的对象
2. m_Name表示对象的名字，这一部分是公用的属性，所有对象都会有
3. m_Obj表示对象本身
4. m_ID表示对象的ID
5. OutputDebugInfo用于输出调试信息

先暂时设计这些属性，后面需要再进行添加；再设置一个枚举，用于区分`_stuObj`的数据类型。

```
//stu类型
enum EType
{
    //角色
    Em_Role,
    //对象
    Em_Object,
    //物品
    Em_Item,
    //地面物品
    Em_GroundItem,
```

```
    //技能
    Em_Skill,
    //已接任务
    Em_GetedTask,
    //可接任务
    Em_CanGetTask,
};
```

OutputDebugInfo函数实现如下：

```
//输出单个对象的调试信息
void _stuObj::OutputDebugInfo()
{
    __try
    {
        switch (m_StuType)
        {
        //输出角色信息
        case Em_Role:
        break;
        //输出对象信息
        case Em_Object:
        break;
        //输出物品信息
        case Em_Item:
        break;
        //输出地面物品信息
        case Em_GroundItem:
        break;
        //输出任务信息
        case Em_GetedTask:
        break;
        //输出技能信息
        case Em_Skill:
        break;
        default:
            break;
        }
    }
    __except (1)
    {
        OutputDebugStringA("输出stuobj信息错误");
    }
}
```

现在我们已经有了一个对象结构体，接着再设计一个对象结构体的集合。

```
//对象结构体集合
struct _stuObjs
{
    //所有的对象集合
    vector<_stuObj> m_data;

public:
```

```
    void OutputDebugInfo();

    //通过名字列表获取对象
    _stuObj GetDataByName(wstring name);

    //通过名字列表获取多个对象
    _stuObjs GetDataByNames(vector<wstring> names);
};
```

`vector<_stuObj> m_data`这个对象结构体的集合用于保存所有的对象。方便后面数据存储。然后再实现三个函数，OutputDebugInfo函数实现如下：

```
//输出多个对象的调试信息
void _stuObjs::OutputDebugInfo()
{
    for (auto it = begin(m_data); it != end(m_data); it++)
    {
        it->OutputDebugInfo();
    }
}
```

由于我们已经实现了单个对象的OutputDebugInfo函数，所以多个对象的OutputDebugInfo只需要调用单个对象的输出函数就可以了。

接着还需要实现一个函数，通过名字获取对象。

```
//通过名字获取对象
_stuObj _stuObjs::GetDataByName(wstring name)
{
    for (auto it = begin(m_data); it != end(m_data); it++)
    {
        if (it->m_Name==name)
        {
            return *it;
        }

    }
    return _stuObj();
}
```

需要实现这个函数的场景是方便我们进行某些功能测试。比如我找到了所有的技能遍历的数据，然后现在需要测试释放技能call，释放技能call一般是传入技能ID，但如果用ID的话，那么我每次都需要去查看我需要释放的ID到底是多少，这样的话就会显得比较麻烦。但如果封装了这个函数就可以直接传入技能名字。

然后再写一个通过名字取多个对象的方法

```
//通过名字列表获取多个对象
_stuObjs _stuObjs::GetDataByNames(vector<wstring> names)
{
    _stuObjs values;
    for (auto it = begin(m_data); it != end(m_data); it++)
    {
        for (auto nit = begin(names); nit != end(names); nit++)
```

```
        {
            if (*nit == it->m_Name)
            {
                values.m_data.push_back(*it);
                break;
            }
        }
    }
    return values;
}
```

CPublic公共函数

这个类存放整个项目都需要用到的公共函数，首先是两个封装好的输出调试信息

```
//*************************************************************
// 函数名称: __OutputDebugStringW
// 函数说明: 打印调试信息
// 作    者: GuiShou
// 时    间: 2020/3/11
// 参    数: pstrFormat
// 返 回 值: void
//*************************************************************
void __OutputDebugStringW(const wchar_t* pstrFormat, ...)
{
    TCHAR szBuffer[1024] = { 0 };
    va_list argList;
    va_start(argList, pstrFormat);
    _vstprintf_s(szBuffer, pstrFormat, argList);
    va_end(argList);
    OutputDebugString(szBuffer);
}


//*************************************************************
// 函数名称: __OutputDebugStringA
// 函数说明: 打印调试信息
// 作    者: GuiShou
// 时    间: 2020/3/11
// 参    数: pstrFormat
// 返 回 值: void
//*************************************************************
void __OutputDebugStringA(const char* pstrFormat, ...)
{
    CHAR szBuffer[1024] = { 0 };
    va_list argList;
    va_start(argList, pstrFormat);
    vsprintf_s(szBuffer, pstrFormat, argList);
    va_end(argList);
    OutputDebugStringA(szBuffer);
}
```

接着，由于x64只有一种调用约定，所以我们可以封装一个函数来对所有的功能call进行调用。例如我们找到的功能call有两个参数时，就可以传入参数直接调用下面的函数

```
QWORD GameCall2(QWORD RCX, QWORD RDX, QWORD calladdr)
{
    __try
    {
        //定义函数指针
        typedef UINT64(*PFnFuncCall)(QWORD RCX, QWORD RDX);

        PFnFuncCall FuncCall = (PFnFuncCall)(calladdr);

        return FuncCall(RCX, RDX);
    }
    __except (1)
    {
        __OutputDebugStringA("通用CALL2  出错\n");
    }
    return 0;
}
```

这样就可以重复利用，而且不用写汇编文件。下面封装的函数也一样，适用于不同参数的call。

```
QWORD GameCall3(QWORD RCX, QWORD RDX, QWORD R8, QWORD calladdr)
{
    __try
    {
        //定义函数指针
        typedef UINT64(*PFnFuncCall)(QWORD RCX, QWORD RDX, QWORD R8);

        PFnFuncCall FuncCall = (PFnFuncCall)(calladdr);

        return FuncCall(RCX, RDX, R8);
    }
    __except (1)
    {
        __OutputDebugStringA("通用CALL3  出错\n");
    }
    return 0;
}


QWORD GameCall4(QWORD RCX, QWORD RDX, QWORD R8, QWORD R9, QWORD calladdr)
{
    __try
    {
        typedef UINT64(*PFnFuncCall)(QWORD RCX, QWORD RDX, QWORD R8, QWORD R9);

        PFnFuncCall FuncCall = (PFnFuncCall)(calladdr);

        return FuncCall(RCX, RDX, R8, R9);
    }
    __except (1)
    {
        __OutputDebugStringA("通用CALL4  出错\n");
    }
    return 0;
}
```

```
QWORD GameCall5(QWORD RCX, QWORD RDX, QWORD R8, QWORD R9, QWORD Rsp20, QWORD
calladdr)
{
    __try
    {
        typedef UINT64(*PFnFuncCall)(QWORD RCX, QWORD RDX, QWORD R8, QWORD R9,
QWORD Rsp20);

        PFnFuncCall FuncCall = (PFnFuncCall)(calladdr);

        return FuncCall(RCX, RDX, R8, R9, Rsp20);
    }
    __except (1)
    {
        __OutputDebugStringA("通用CALL5  出错\n");
    }
    return 0;
}

QWORD GameCall6(QWORD RCX, QWORD RDX, QWORD R8, QWORD R9, QWORD Rsp20, QWORD
Rsp28, QWORD calladdr)
{
    __try
    {
        typedef UINT64(*PFnFuncCall)(QWORD RCX, QWORD RDX, QWORD R8, QWORD R9,
QWORD Rsp20, QWORD Rsp28);

        PFnFuncCall FuncCall = (PFnFuncCall)(calladdr);

        return FuncCall(RCX, RDX, R8, R9, Rsp20, Rsp28);
    }
    __except (1)
    {
        __OutputDebugStringA("通用CALL6  出错\n");
    }
    return 0;
}
```

MemOper 内存操作

这个类封装所有数据类型的取内容操作，目的是提高代码可读性

```
#include "pch.h"
#include "MemOper.h"


BYTE ReadBYTE(QWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(BYTE)) == 0)
        {
            return *(BYTE*)pBase;
        }
    }
    __except(1)
    {
```

```
        __OutputDebugStringA("ReadBYTE Error Addr:%x", pBase);
    }

    return 0;
}

BYTE ReadBYTE(DWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(BYTE)) == 0)
        {
            return *(BYTE*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadBYTE Error Addr:%x", pBase);
    }
    return 0;
}

WORD ReadWord(QWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(WORD)) == 0)
        {
            return *(WORD*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadWord Error Addr:%x", pBase);
    }

    return 0;
}

WORD ReadWord(DWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(WORD)) == 0)
        {
            return *(WORD*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadWord Error Addr:%x", pBase);
    }

    return 0;
}

QWORD ReadQword(QWORD pBase)
{
```

```
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(QWORD)) == 0)
        {
            return *(QWORD*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadQword Error Addr:%x", pBase);
    }

    return 0;
}


DWORD ReadDword(QWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(QWORD)) == 0)
        {
            return *(DWORD*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadDword Error Addr:%x", pBase);
        return 0;
    }



    return 0;
}

DWORD ReadDword(DWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(DWORD)) == 0)
        {
            return *(DWORD*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadDword Error Addr:%x", pBase);
        return 0;
    }

    return 0;
}



int ReadInt(QWORD pBase)
```

```
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(int)) == 0)
        {
            return *(int*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadInt Error Addr:%x", pBase);
    }

    return 0;
}

float ReadFloat(QWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(float)) == 0)
        {
            return *(float*)pBase;
        }
    }
    __except(1)
    {
        __OutputDebugStringA("ReadFloat Error Addr:%x", pBase);
    }

    return 0;
}


float ReadFloat(DWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(float)) == 0)
        {
            return *(float*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadFloat Error Addr:%x", pBase);
    }

    return 0;
}

void WriteFloat(QWORD pBase,float Value)
{
    __try
    {
        if (IsBadWritePtr((void*)pBase, sizeof(float)) == 0)
        {
            *(float*)pBase = Value;
```

```
        }
    }
    __except (1)
    {
        __OutputDebugStringA("WriteFloat Error Addr:%x", pBase);
    }


}


wchar_t* ReadWChar(QWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(wchar_t)) == 0)
        {
            return (wchar_t*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("WriteFloat Error Addr:%x", pBase);
    }

    return L"";
}

char* ReadChar(QWORD pBase)
{
    __try
    {
        if (IsBadReadPtr((void*)pBase, sizeof(char)) == 0)
        {
            return (char*)pBase;
        }
    }
    __except (1)
    {
        __OutputDebugStringA("ReadChar Error Addr:%x", pBase);
    }

    return "";
}
```

GlobalData 全局数据

这个里面存放游戏需要用到的全局数据

.h文件

```
//全局的游戏模块地址
extern QWORD g_GameAddr;
```

.cpp文件

```
QWORD g_GameAddr = 0;

void GetGameModuleAddr()
{
    g_GameAddr = (QWORD)GetModuleHandleA("MMOGame-Win64-Shipping.exe");
}
```

然后在界面初始化的时候调用一次，后续就可以一直使用游戏的模块基地址了。

剩下的GameData和GameFunction模块存放游戏数据代码和游戏功能代码，我们等需要用到的时候在添加。