# 程序数据跟踪的讨论

nEINEI/2006-03-30

前一段时间在一个技术论坛讨论过，什么样的程序 BUG 最难调试 。 总结后大概有以下几个观点。
1 程序运行一段时间后崩溃。
2 程序退出时崩溃。
3 程序运行中偶尔崩溃，且不容易复现
4 程序启动时出错。
5 系统资源暗自消耗。
6 多线程处理
7 DEBUG 版本下没有问题而在 release 版本下出现的错误

本文所说的程序数据跟踪就是要跟踪到这些难于调试的 BUG 中。

一 ，难调试 BUG 产生的原因

从以上几个观点中我们可以看出之所以难以调试，在于不容易定位到出错的代码位置，不容易准确定位在于无法得到出错位置的必要信息。相比容易调试的 BUG 来说，运行一段时间崩溃的程序使我们不容易猜到出错的位置，当然可以凭借经验来定位出错的位置。但这却是耗时耗力的事情。

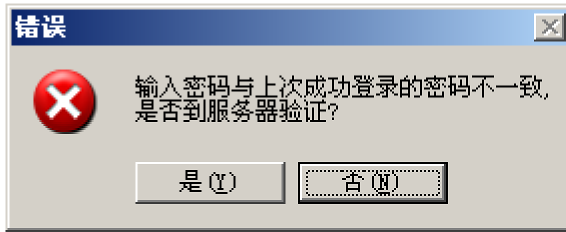
不容易定位的根本原因在于缺少相应的方法将程序的最后状态输出出来。而且这种错误不是在编译期间产生的，又无法在调试状态跟踪到。如果能得到程序非正常结束时的运行状态，这样就可以快速找到出问题的地方。 如何能得到这样的状态呢？让我们先对比一下容易调试的 BUG 与不容易调试的 BUG 的具体区别，看看是什么阻挡了我们调试的进行，并以此寻找新的方法。

二 ， 容易调试 BUG 与难调试 BUG 的对比

区别 1 有明显的现象。容易调试的 BUG 很容易露出狐狸的尾巴，比如 读写注册表操作，
读写文件操作，成功或者失败发生现象都是明显的。这些信息都给定位出错位置提供最直接的支持。
区别 2 有相应的程序提供出错处理。 BUG 的产生虽然有多种可能，但在程序中我们都通过严格编写规则对出错部分作出错处理，这样若出错处理部分未按我们的预计情况执行则可以马上判断出 BUG 位置。
例如 if ( 条件 == 真 ) （当然这种写法并不好，应该是 if(条件) 在这只是为说明问题）

```
{
     继续处理 ；
}
 Else
 {
     可能是另一个处理分支，也可能出错处理部分 ，
     在这里提示用户其所做的操作不被程序认可。
}
```

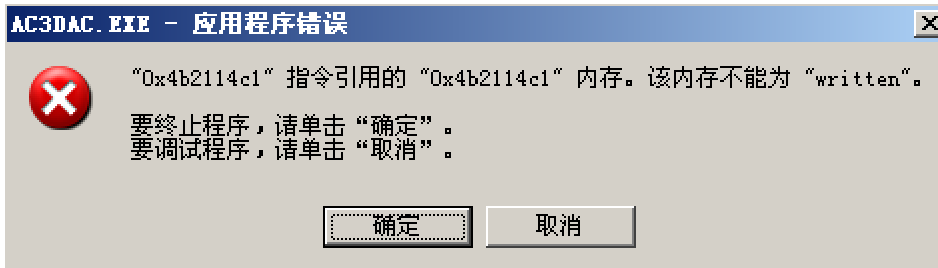
例如运行输入口令的程序，当口令错误时程序会给我们提示。




类似的处理方法还有利用 ASSERT 宏 。

区别 3 有反映程序状态的提示信息 。 这些信息都是用来提示用户或引导用户正确操作的“帮助”信息，那么在此之间出现的错误，都可以通过这些信息来定位我们程序出错的位置。这些信息无形中也帮助了我们自己。

区别 4 缺少有用信息的提示 。不容易调试的 BUG 大都会有类似下面的提示，
这一基于保护模式的操作系统给了我们程序最严重的警告。可这提示对我们调试程序的帮助确又不大。




通过以上的对比我们可以发现，这两类 BUG 的不同特点。那就是一个在明处，一个在暗处 ，一个出了事可以很快找到它，一个则需要我们自己细细地排查。由此我们想到要能把不容易调试的 BUG 转换成容易调试的 BUG 那问题就解决了。要解决的有两点 1 要有明确的提示，2 记录程序运行的状态 。

三 ， 程序数据跟踪的产生

TRACE 是我们在调试状态下经常使用的宏 。它将我们输出的必要信息输出到控制台中，用于观察具体是数据。这些数据对用户来说是透明的，借助于这个想法，如果我们能将程序运行中可能出错的数据及程序的状态输出的一个介质中，并利用相应的工具查看这些数据，那么我们就会得到十分有用的信息，并且可以快速地定位到 BUG 处 。本文所讨论的数据跟踪就时基于这一想法实现的最简单的 Track。

笔者为此编写了跟踪数据的类 Track ，和用于查看数据的工具 TrackView 。
先看一个应用 ，下图是一个程序启动失败的提示，此时已应用了 Track，在这个程序中导致初始化失败的原因有多种可能，单从初始化失败是不容易判断出问题的所在位置的。

利用 TrackView 来查看情况如图。

通过最后的输出 SignatureCheck 可以快速定位出程序在 InitInstance 中调用 SignatureCheck 函数时失败，在该函数中验证 MD5 失败。这样远比跟踪调试快得多。关键是对运行过程中偶尔出错程序提供了数据跟踪，这样可以有效的帮助我们定位到出错位置。下面对 Track 类进行详细的说明。

四 ， 数据跟踪类 Track 的实现

首先 Track 要有输出功能的函数，OutPut 用于将数据输出到介质，可以是内存也可以是文件中，在这里选择了文件 。对于数据则主要是字符和数字。所以重载 OutPut 函数 。定义如下：

```
bool Output(LPCTSTR lpCaption   PCTSTR pData   LPCTSTR Type = NULL, bool b = false);
bool Output(LPCTSTR lpCaption , int nData,          LPCTSTR Type = NULL, bool b = false);
```

lpCaption ----- 特征标题，用于说明函数名或其它特殊说明。
pData -------- 输出的数据
nData -------- 输出的数据
Type -------- 输出的数据类型，默认是 NULL
Bool b -------- 是否加特殊标记，当产生数据很多时，可以通过使用特殊标记加快找到所关心的数据。

因为 Track 会在程序运行时产生文件，而发布的版本是不能含有调试信息的。所以还应该有一个调试状态的控制开关 。

```
SetDebug (bool  b )  ;
```

// b---- true 调试状态，产生文件，b------false 非调试状态，不产生文件

Track 定义如下：

```
class CTrack
{
public:
	CTrack();
	CTrack(LPCTSTR lpParam);
	virtual ~CTrack();
	void SetDebug(bool b) ;
	bool Output(LPCTSTR lpCaption,LPCTSTR lpData,LPCTSTR Type = NULL,bool b = false);
	bool Output(LPCTSTR lpCaption ,int   nData,LPCTSTR Type = NULL,bool b = false);
	public :
	CStdioFile m_File;
	bool m_b;                    // 判断文件是否可读写
	bool m_bSpecial;             // 判断是否加特殊标识
	CString m_strPathName;       // 生成路径
	bool m_bAuto;                // 自动生成在程序路径
	bool m_bOutput ;             // 判断是否在调试状态
private :
	void SetTrack(LPCTSTR lpPathName = NULL);
	void GetCurPath();

};
```

相应实现：

```
CTrack::CTrack()
{
	m_bOutput = true ;
	m_b = false;
	m_bAuto = false ;
	GetCurPath();
	SetTrack();
}
CTrack::CTrack(LPCTSTR lpParam)
{
		m_bOutput = true ;
		m_b = false;
		m_bAuto = false ;
		GetCurPath();
		SetTrack(lpParam);
}
bool CTrack::Output(LPCTSTR lpCaption,LPCTSTR lpData,LPCTSTR Type,bool b)
{
	if (m_bOutput)
```

```
{
	CString str;
	str.Format("%s",lpCaption);
	str += "*";
	str += lpData;
	str += "*";

	m_bSpecial = b;

	if (Type != NULL)
	{
		str += Type;
	}
	else
	{
		str += "NULL";
	}

	str += "*";

	if (b != NULL)
	{
		str += "有特殊标记";
	}
	else
	{
		str += "NULL";
	}

	if(m_b)
	{

		if (m_File.Open(m_strPathName,CFile::modeReadWrite))
		{
			char szBuffer[MAX_PATH] = {0};
			sprintf(szBuffer,"%s",str);
			m_File.SeekToEnd();
			m_File.WriteString(str+'\n');
			m_File.Close();
		}
	}
	return 0 ;
}
return 0 ;
```

```
}
bool CTrack::Output(LPCTSTR lpCaption ,int nData,LPCTSTR Type,bool b)  与上一个类似故省略。

void CTrack::SetTrack(LPCTSTR lpPathName)
{
	if (lpPathName != NULL)
	{
		m_strPathName.Format(lpPathName);
		CFileFind ff;
		int	nRet = ff.FindFile(lpPathName);
		ff.Close();
		if(nRet != 0)
		{
			if(!m_File.Open(lpPathName,CFile::modeReadWrite))
			{
				MessageBox(NULL,"Track  无法读写您指定路径下的文件,请检查后运行或更换其他
文件","Track  提示",MB_OK);
			}
			else
			{
				m_b = true;
				m_File.Close();
			}
		}
		else
		{
			if(!m_File.Open(lpPathName,CFile::modeCreate | CFile::modeReadWrite))
			{
				MessageBox(NULL,"Track  无法打开您指定路径下的文件,请检查后运行或更换其他
文件","Track  提示",MB_OK);
			}
			else
			{	m_b = true;
				m_File.Close();
			}
		}
	}
	else
	{
		CFileFind ff;
		int	nRet = ff.FindFile(m_strPathName);
		ff.Close();
		if(nRet != 0)
		{
```

```
			if(!m_File.Open(m_strPathName,CFile::modeReadWrite))
			{
				MessageBox(NULL,"Track 无法读写您指定路径下的文件,请检查后运行或更换其他
文件","Track 提示",MB_OK);
			}
			else
			{
				m_b = true;
				m_File.Close();
			}
		}
		else
		{
			if(!m_File.Open(m_strPathName,CFile::modeCreate | CFile::modeReadWrite))
			{
				MessageBox(NULL,"Track 无法打开您指定路径下的文件,请检查后运行或更换其他
文件","Track 提示",MB_OK);
			}
			else
			{
				m_b = true;
				m_File.Close();
			}
		}
	}

}

void CTrack::GetCurPath()
{
	CString str ;
	char szFileName[MAX_PATH] = {0};
	GetModuleFileName(NULL , szFileName , sizeof(szFileName));
	str.Format("%s",szFileName);
	int pos = str.ReverseFind('\\');
	m_strPathName = str.Left(pos);
	m_strPathName += "\\Track.trc";
	CRegKey reg;
	BOOL b ;
	b = reg.Open(HKEY_LOCAL_MACHINE,"SOFTWARE\\Track");
	if (b == ERROR_SUCCESS)
	{
		reg.SetValue(m_strPathName,"PATH");
	}
```

```
	else
	{
		if (reg.Create(HKEY_LOCAL_MACHINE,"SOFTWARE\\Track"))
		{
			reg.SetValue(m_strPathName,"PATH");
		}
	}
	reg.Close();
}
void CTrack::SetDebug(bool b)
{
	m_bOutput = b ;
	CFileFind ff;
	if (!b)
	{
		if (ff.FindFile(m_strPathName))
		{
			DeleteFile(m_strPathName);
		}
		ff.Close();
	}
}
```

这样最基本的 Track 就被我们构造好了， 我们看一下如何使用 。
因为把声明和实现都写在了 Track.h 文件中了。只需将该文件放入
C:\Program Files\Microsoft Visual Studio\VC98\Include\Track.h 或
C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\include\Track.h 即可

在使用处加入

```
#include "Track.h"

CTrack  myTrack;

myTrack.SetDebug(true) ;   // 调试状态，发布程序时要设置成 SetDebug（false）

BOOL CLXApp::InitInstance()
{
	myTrack.OutPut( “InitInstance” ,”  进入” );
	..........

	..........
	if(!gSign->SignatureCheck())
	{
```

```
                myTrack.Output("SignatureCheck","失败",0,true);
                ..........
        }
}

bool LXSignature::InitSignList()
{
        ………
        if(!DecryptFile(strSrcFile, pTargetBuf, dwSize))
        {
                myTrack.Output("DecryptFile","解签失败",0,true) ;
                return false;
        }
     ………..
}

bool LXSignature::SignatureCheck()
{
        ………
if(strFileMD5 != strMD5)
{
        myTrack.Output("SignatureCheck","验证MD5失败",0,true) ;
        return false;
}
        ………
}
```

这样程序运行过程中的关键状态都会被我们记录下来，可能大家会感到疑惑为什么不在关键状态处弹出 MessageBox 呢，它的作用和 Track 一样啊，可仔细一想就知道不行，第一 程序运行过程的验证机制对用户是透明的，程序的内部流程也是不想被人知道的 。第二 程序中有那么多的 MessageBox 也是比较烦的事情。

五，关于未来 Track 的想法

通过 Track 我们记录了程序的关键状态，达到了跟踪作用，对程序运行中偶尔崩溃，且不容易复现程序也可以发现其蛛丝马迹 。但 Track 却并不完善，它同样依赖于程序员的经验，对可能出错的地方加入 OutPut(….) ，这毕竟不是很方便的事情，智能的 Track 应该有一套自动化的机制，应该在源代码层面进行分析，程序员并不需要主动参与数据跟踪，Track 能主动输出有效数据，基于这样思想的 Track 将是程序员调试 BUG 的利器。

结束语

我们写的程序都具有很高的安全性，每个产品都有相应的验证机制，数字签名机制，硬件指纹检测等手段而且一个项目或产品又都划分为多个模块，每位程序员负责不同的模块编写工作，这样对于其它同事的程序编写情况并不十分了解，在程序运行过程中产生的问题就不容易定位,到底是谁负责的模块出了问题呢。当找到出问题的地方时可能已经花费了很多的时间了，这样既浪费了我们的时间又花费了我们很大的精力，关键是并不会因为我们这次找到了错误，下次就不发生了。Track 为我们提供了一个手段，使我们

能够掌控程序的运行状态，跟踪到出错的位置，对快速定位程序 BUG 起到了一定的帮助作用，也节省了时间。