PbIdea 应用之：PowerBuilder 里如何调用 DLL 函数

看到这个标题，许多人感觉用 PB 调用第三方 DLL 函数，这还不是很简单的事嘛，手到擒来！

事实真的是这么简单吗？下面我们看几个例子。

例一：

DllTest.dll 里导出函数：

```
extern "C" __declspec(dllexport) int ReturnRefA(const char* text,char* ref,int* len);
```

PB 里写声明

```
Function long ReturnRefA(ref string strText,ref string strRef,ref long nRef) library "DllTest.dll"
```

接下来，我们调用它：

```
String ls_text,ls_ref
Long ll_long
Ls_text = space(100); ls_ref = space(100)
ReturnRef(ref  ls_text,ref  ls_ref,ref  ll_len)
```

运行它！结果，你会发现，程序会报错，弹出一个对话框，内容是

“Specified argument type differs from required argument type at runtime in DLL function name”，意思就是"指定的参数类型与动态链接库中的函数所需要的参数类型不一致"。这是什么原因？原来，我们的导出函数没有使用 __stdcall（WINAPI） 调用约定，上面这个 DLL 导出函数如果写成：

```
extern "C" __declspec(dllexport) int __stdcall ReturnRefA(const char* text,char* ref,int* len);
```

并且把 ReturnRefA 这个函数名添加到 DllTest.def 里，再编译成 DLL，上面的 PB 代码就可以正常跑了。

然而，许多第三方 DLL 导出函数，并不没有使用__stdcall 调用约定，而是使用了默认的__cdecl 调用约定（不写约定就默认为__cdecl），我们的 PB 当然无法直接调用了。

例二：

我们来看一个 windows 的 WINAPI 函数：

```
BOOL WINAPI EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

这个函数有 2 个参数，第二个参数，我们知道是一个 LONG 型，这个好办，那么，第 1 个参数是什么东西？百度搜索，原来它是 BOOL CALLBACK EnumWindowsProc(_In_ HWND hwnd,_In_ LPARAM lParam);这么一个东西。这是啥啊？怎么又以是一个函数啊？

这就是 callback，即回调。回调是一个指向函数所在地址的指针。这点咱们 PB 就无能为力了。

例三：

其他一些特殊之处。我们知道 PB9 是 ansi 编码，PB10 以上是 unicode 编码，那么如果在 PB9 里处理 DLL 函数里使用到 unicode 编码，在 PB10 里 DLL 函数的 ansi 编码，就不那么方便了，虽然 PB10 以上可以用 ;ansi 支持，但有时候也不是那么灵光。

以上问题，咱们 pbidea 库里，使用 uo_dll 对象，都可以轻松解决。

在接下来的内容之前，建议阅读一下我的另一篇内容《PowerBuilder 中调用 DLL 参数类型》https://blog.csdn.net/lxbin2003/article/details/121527801。

这里，我们先把“神秘”的“指针”给简单介绍一下。所谓指针，其实就是内存里指向某个位置的一个表示，它以一个数字表示，在 PB 里就是一个 ULONG，相当于是一个 ULONG 类型的数字标识出在某个位置的那块内存，所以在 PB 里，我们可以把所有的指针，就理解为一个 ULONG 类型的数字。至于它指向的那块内存里是什么，这个由应用决定：它可以是一个字符串，可以是 BLOB 数据，可以是 LONG、 INT 等各种数据，甚至是它们的一个组合体（结构）。简单点：**我们指针理解为一个 ULONG 类型的数字就行了**。

现在，我们开始了解 PbIdea 之 uo_dll 是如何工作的。

一、获取 DLL 里的函数列表

```
//取函数列表
function int LoadFunctionList(readonly string dllName,ref string funcNameList[]) system
```

library "pbidea.dll" alias for "funcLoadFunctionList"

function int LoadFunctionList(readonly string dllName,ref string funcNameList[],ref string addr[]) system library "pbidea.dll" alias for "funcLoadFunctionList"

使用代码：

```
string funcs[],addr[]
uo_dll dll
dll = create uo_dll
dll.loadfunctionlist(is_dll,ref funcs[],ref addr[])
int i
for i = 1 to UpperBound(funcs)
  dw_1.InsertRow(0)
  dw_1.SetItem(i,1,funcs[i])
  dw_1.SetItem(i,2,addr[i])
next
```

测试用 DllTest.dll 里的函数列表

Funcname	Address
ReturnA	55921220
ReturnRefA	55921260
ReturnRefW	559212F0
ReturnW	55921240
StartAsyncCallback	559217B0
StopAsyncCallback	55921920
TestA	559211C0
TestNotExport	559213A0
TestW	559211F0
_StdCallFunction@8	559213C0

这样就把函数列表取出来了。使用方法其实很简单。这个就省得去找其他工具观察 DLL 里的函数了，直接就可以看到。

二、从一个简单的函数调用开始

DLL 导出函数：extern "C" __declspec(dllexport) int TestA(const char* text)

这是一个 ansi 字符串编码函数，一个参数。以下结合调用代码详细解释。

```
uo_dll dll;dll = create uo_dll //创建对象实例
dll.LoadFunction(is_dll, "TestA",dll.CDECL) //is_dll 是一个实例变量 string is_dll = "DllTest.dll"。加载函数。
string ls_text
ls_text = "我是 ansi 编码字符串"
dll.PushArg(ls_text,dll.ANSI) //添加一个参数
long ret
if dll.Invoke(ret) then //调用
  messagebox("返回值是:",string(ret))
end if
destroy dll
```

以上代码并不复杂，用到 3 个函数。

1. LoadFunction：加载函数。

函数声明：function boolean LoadFunction(readonly string dllName,readonly string

funcName,int mode) system library "pbidea.dll" alias for "funcLoadFunction"

这个函数有 3 个参数，分别是：DLL 名称，函数名称，调用约定(CDECL/STDCALL,)

调用约定的定义是：

```
//调用约定
constant int CDECL   = 1
constant int STDCALL = 2
```

dll.LoadFunction(is_dll, "TestA",dll.CDECL) 这句意思就是从 dlltest.dll 里用 CDECL 调用约定加载 TestA 这个函数，返回 true/false，表示是否加载成功。

2.PushArg：添加一个参数。

函数声明：

```
function boolean PushArg(readonly boolean  value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly char     value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly int      value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly uint     value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly long     value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly ulong    value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly longlong value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly real     value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly decimal  value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly double   value) system library "pbidea.dll" alias for "funcPushArg"
//字符编码 charset: 0 默认当前编码，1 转 ansi 编码，2 转 unicode 编码
function boolean PushArg(readonly string   value) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly string     value, int charset) system library "pbidea.dll" alias for "funcPushArg"
function boolean PushArg(readonly blob     value) system library "pbidea.dll" alias for "funcPushArg"
```

这个函数有多个重载，适配不同参数类型。当参数类型是 string 时，注意是 2 个重载。如果直接使用 string 参数，会以当前编码方式添加参数，即 PB9 就是 ansi，PB10 以上是 UNICODE。还有种重载是 2 个参数，第二个参数可以指定转成什么编码方式。

编码方式的定义是：

```
//编码方式
constant int ANSI    = 1
constant int UNICODE = 2
```

返回 true/false，表示是否成功。

3. Invoke：通过声明、添加参数，做好了准备，接下来就是调用函数。

function boolean Invoke(ref boolean returnValue) system library "pbidea.dll" alias for "funcInvoke"

function boolean Invoke(ref int returnValue) system library "pbidea.dll" alias for "funcInvoke"

function boolean Invoke(ref long returnValue) system library "pbidea.dll" alias for "funcInvoke"

function boolean Invoke(ref longlong returnValue) system library "pbidea.dll" alias for "funcInvoke"

function boolean Invoke(ref real returnValue) system library "pbidea.dll" alias for "funcInvoke"

function boolean Invoke(ref double returnValue) system library "pbidea.dll" alias for "funcInvoke"

这是函数声明，有个重载，它有一个参数，用于返回函数调用的返回值。比如 ReturnRefA 返回一个 int 类型，C 里的 int，PB 里是 long.所以

long ret; dll.Invoke(ret) 这样来接收返回值。

Invoke 返回 true/false，表示是否调用成功。

三、如何获取 DLL 函数的返回字符串或者内存内容

```
//调用函数
function boolean Invoke(ref boolean  returnValue) system
function boolean Invoke(ref int      returnValue) system
function boolean Invoke(ref long     returnValue) system
function boolean Invoke(ref longlong returnValue) system
function boolean Invoke(ref real     returnValue) system
function boolean Invoke(ref double   returnValue) system
```

我们看到，invoke 这个函数，它的引用参数里只是一个普通基本类型，没有 string 这一个类型，那么，函数如果返回的是字符串，那么我们如何接收呢？

还记得上面我们提到过：**我们指针理解为一个 ULONG 类型的数字就行了**吗？

是了，就用 long 或 ulong 接收它。

下面看看调用 DLL 导出函数 extern "C" __declspec(dllexport) const char* ReturnA(const char* text)的方法：

```
uo_dll dll
dll = create uo_dll
dll.LoadFunction(is_dll, "ReturnA",dll.CDECL)
string ls_text,ls_ret
ls_text = "我是 ansi 编码字符串"
dll.PushArg(ls_text,dll.ANSI)
long ret
if dll.Invoke(ret) then
   //返回的 ret 是指向 ansi 字符串的地址
   ls_ret = dll.ReadAnsiString(ret)
   messagebox("返回值是:",ls_ret)
end if
destroy dll
```

使用 long ret ，接收一个返回地址，这个地址指向字符串内容。然后使用 ReadAnsiString 这个函数，从这个地址读取字符串内容即可。

读字符串内容有 3 个函数，看声明和注解，就可以理解它们的用处。

//读内存字符串，自动转码

function string ReadString(ulong mem) system library "pbidea.dll" alias for "funcReadString"

//读 ansi 编码内存字符串，自动转码

function string ReadAnsiString(ulong mem) system library "pbidea.dll" alias for

"funcReadAnsiString"

//读 unicode 编码内存字符串,自动转码

function string ReadUnicodeString(ulong mem) system library "pbidea.dll" alias for "funcReadUnicodeString"

思考：

如果我们添加一个 UNICODE 字符串作为参数，接收一个返回 UNICODE 的字符串，该如何修改以上代码呢？

那么添加一个 ansi 字符串作为参数，接收一个返回 UNICODE 的字符串，该如何修改以上代码呢？

或者那么添加一个 UNICODE 字符串作为参数，接收一个返回 ANSI 的字符串，该如何修改以上代码呢？

四、使用输出型参数

许多 DLL 函数有输出参数,用 PB 的概念来说就是引用型参数,这类参数可以接受 DLL 函数的输出结果。

PB 调用引用类型参数，首先要为这个参数分配一块空间，通常我们用 space(…)这样来操作。实际上，任何语言使用输出类型参数，都要预先分配足够大空间。

DLL 导出函数 extern "C" __declspec(dllexport) int ReturnRefA(const char* text,char* ref,int* len) 调用实例

这个函数有 1 个入参，2 个输出参数。注意 C 里的 int 对应 PB 的 long，这点很重要。

```
uo_dll dll
dll = create uo_dll
dll.LoadFunction(is_dll, "ReturnRefA",dll.CDECL)
string ls_text,ls_ret
long ret,ret_ref,ret_len,ll_ret
ls_text = "我是 ansi 编码字符串"
dll.PushArg(ls_text,dll.ANSI)
//为返回字符串分配空间,地址保存起来备用
ret_ref = dll.PushRefArg(512)
//为返回 int 分配空间, 4个字节就够了,地址保存起来备用
ret_len = dll.PushRefArg(4)
if dll.Invoke(ret) then
   //返回的 ret 是指向 ansi 字符串的地址
   ls_ret = dll.ReadAnsiString(ret_ref)
   ll_ret = dll.ReadLong(ret_len)
   messagebox("返回值是:",ls_ret + " , 长度:" + string(ll
end if
dll.FreeMemory(ret_ref) //释放内存
dll.FreeMemory(ret_len) //释放内存
destroy dll
```

这段代码里，我们使用了 2 个新函数:

//添加引用参数，返回分配的内存地址

function ulong PushRefArg(long bufferSize) system library "pbidea.dll" alias for "funcPushRefArg"

//读一个 long

function long ReadLong(ulong mem) system library "pbidea.dll" alias for "funcReadLong"

PushRefArg 会分配一个指定大小的以字节为单位的内存块作为参数，并返回这个地址，这个地址要保存到变量里，以备后面取参数，并且释放使用。

ReadLong 是从指定内存位置读一个 long 类型，注意：long 是 4 个字节。另外还有个函数：

//读一个 int

function int ReadInt(ulong mem) system library "pbidea.dll" alias for "funcReadInt"

ReadInt 是从指定内存位置读一个 int 类型，注意：int 是 2 个字节。

ret_ref = dll.PushRefArg(512) 是添加 512 个字节作为返回字符串空间
ret_len = dll.PushRefArg(4) 是添加 4 个字节作为 long 类型的返回内存空间.
函数调用成功后
  ls_ret = dll.ReadAnsiString(ret_ref)
  ll_ret = dll.ReadLong(ret_len)
获取返回结果。
最后
dll.FreeMemory(ret_ref) //释放内存
dll.FreeMemory(ret_len) //释放内存
把分配的内存空间释放掉。

```
//为返回字符串分配空间,地址保存起来备用
ret_ref = dll.PushRefArg(512)
//为返回 int 分配空间, 4个字节就够了,地址保存起来备用
ret_len = dll.PushRefArg(4)
```

这里添加引用参数，是由 uo_dll 分配了内存，还有另外一个方法，即我们自己分配内存，然后把分配的内存作为参数传给它。这里用到另一个函数：

//分配内存
function ulong AllocMemory(long sizeOfMemory) system library "pbidea.dll" alias for "funcAllocMemory"
我们还可以这样写：
Ret_ref = dll. AllocMemory(512)
Ret_len = dll. AllocMemory(4)
Dll.pushArg(ret_ref)
Dll.pushArg(ret_len)
效果和以上一样。

那么如果不是字符串，而是一个结构呢？
结构大小由我们自己计算，如果不会计算，就尽量给大点，使用 AllocMemory 分配大块内存，总不会有错。
//读指定大小的内存块
function blob ReadMemory(ulong mem, long len) system library "pbidea.dll" alias for "funcReadMemory"
然后使用 ReadMemory 函数，从指定内存里读取指定字节数，得到一个 blob，然后自己处理这个 blob。

//复制内存块, 注意，此函数参数可以根据需要，任意改变 !!!
subroutine CopyMemory(ulong dst,ulong src,long size) library "pbidea.dll" alias for "funcCopyMemory"
CopyMemory 函数可以在不同数据类型，或者结构间互相复制。这个函数需要根据业务需要，自己做更多声明。
例如：
subroutine CopyBlobToStruct(ref str_people dst,ref blob src,long size) library "pbidea.dll" alias for "funcCopyMemory"
这样就可以把一个 blob 内容复制到一个 struct 里面去了。

五、线程内同步回调

我们使用 WINAPI 的 EnumWindows 举例。这个函数它的调用 和回调，都发生在同一个线程里，比较简

单。先看 MSDN 的声明：

BOOL WINAPI EnumWindows([in] WNDENUMPROC lpEnumFunc,[in] LPARAM lParam);

BOOL CALLBACK EnumWindowsProc(_In_ HWND hwnd,_In_ LPARAM lParam);

EnumWindows 函数的作用是枚举所有窗口。它有 2 个参数，第一个参数是一个回调，第二个参数是自定义参数。通常我们使用这个参数传一个字符串，在回调用取窗口的标题，与这个字符串比较，从而判断这个窗口是不是我们需要查找的那个窗口。

EnumWindowsProc 就回调的具体实例。在这里，我们另外定义它的实现形式。

```
string ls_text
ls_text = "pbidea" //自定义参数
dw_2.reset()

uo_dll dll
dll = create uo_dll
//声明函数
dll.LoadFunction("user32.dll", "EnumWindows",dll.STDCALL)

//创建回调参数,参考EnumWindowsProc声明
int cb
cb = dll.CreateCallback(dll.STDCALL,dll.BOOL_TYPE)
dll.PushCallbackArgType(cb,dll.ULONG_TYPE)
dll.PushCallbackArgType(cb,dll.ULONG_TYPE)

//给函数添加2个参数,参考EnumWindows 声明
dll.PushCallbackArg(cb)
dll.PushArg(ls_text)

//绑定一个回调事件
dll.BindCallbackEvent(cb,parent,"ue_callback")

//调用，并返回 boolean 类型
boolean ret
if dll.Invoke(ret) then
   messagebox("返回值是:",ret)
end if

destroy dll
```

注意：dll.LoadFunction("user32.dll", "EnumWindows",dll.STDCALL) 这里使用的是 STDCALL 调用约定。WINAPI 默认都是 STDCALL 调用约定。

这里使用了几新函数：

//返回 标地号，其他相关 callback 类函数使用此 ID.

function int CreateCallback(int mode,int returnType) system library "pbidea.dll" alias for "funcCreateCallback"

//添加回调参数

function boolean PushCallbackArg(int callback) system library "pbidea.dll" alias for "funcPushCallbackArg"

//给回调添加参数类型,具体 类型参考声明

function boolean PushCallbackArgType(int callback,int argType) system library "pbidea.dll" alias for "funcPushCallbackArgType"

//将回调与事件进行绑定

function boolean BindCallbackEvent(int callback,powerobject obj,readonly string eventname) system library "pbidea.dll" alias for "funcBindCallbackEvent"

cb = dll.CreateCallback(dll.STDCALL,dll.BOOL_TYPE)：创建一个回调接口，实际上就是 EnumWindowsProc 的一个实现，返回一个 int 类型的值，这个值就唯一指定这个回调。注意这个回调的调用

约定是 STDCALL,看 MSDN 的声明“BOOL CALLBACK EnumWindowsProc”，这里面的“CALLBACK”实际上就是 STDCALL 约定。它的第 2 个参数指明了这个回调的返回值类型。

dll.PushCallbackArgType(cb,dll.ULONG_TYPE)

dll.PushCallbackArgType(cb,dll.ULONG_TYPE)

给这个回调绑定 2 个参数，参数类型都指定为 ULONG_TYPE。分别指向 HWND hwnd,_In_ LPARAM lParam。

至此，一个回调就构建好了。

然后

dll.PushCallbackArg(cb) //把回调作为一个参数传递给 EnumWindows 函数

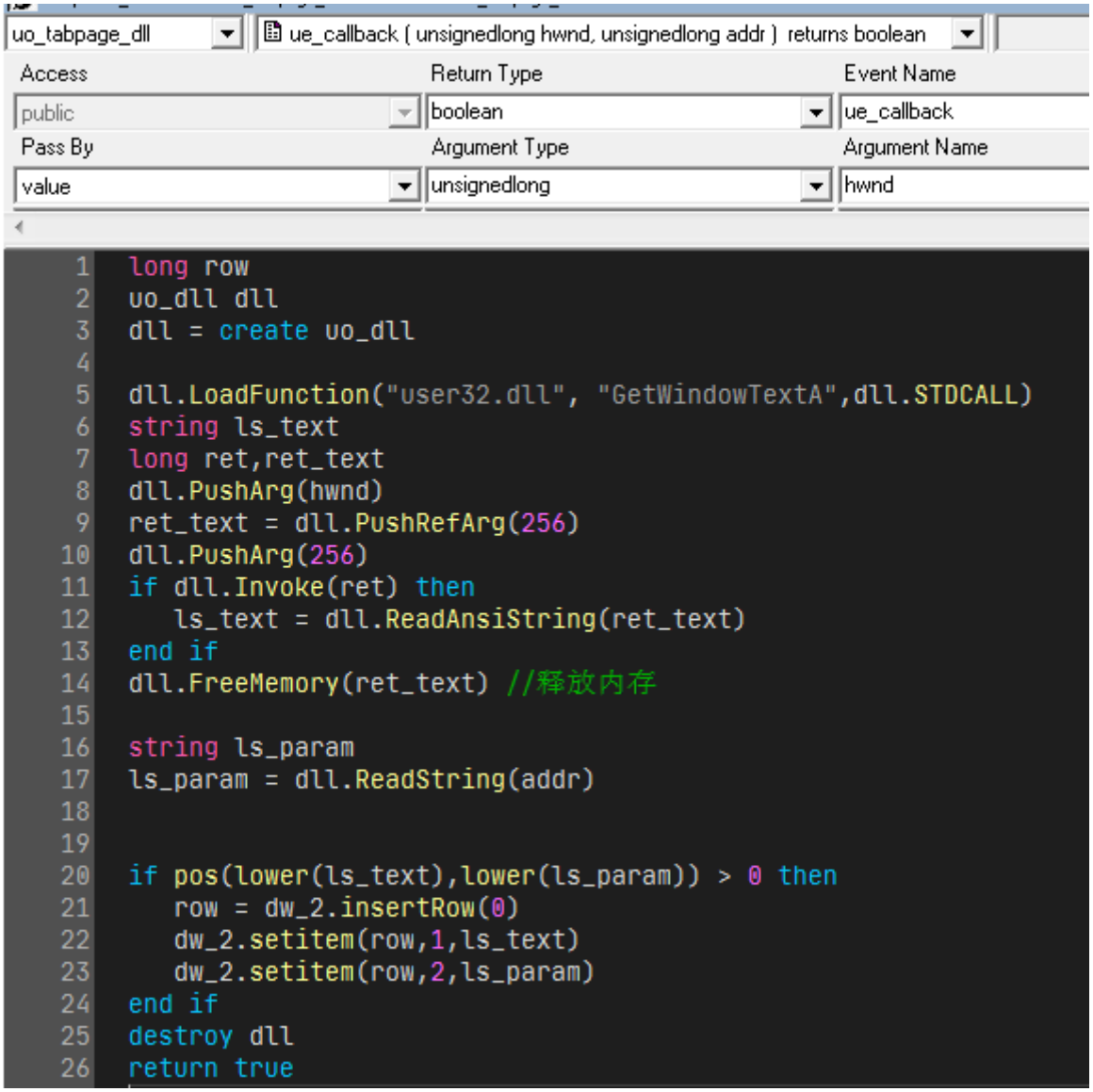
dll.PushArg(ls_text)　　// 传递一个自定义参数

接下来很重要：

dll.BindCallbackEvent(cb,parent,"ue_callback") ，把一个回调绑定到一个对象实例的事件上，这样当这个回调发生时，会触发这个事件，我们在这个事件里去实现我们的回调业务。这个绑定内部有检查，如果绑定的事件声明与创建回调时指定的参数类型不一致时，会报错，函数会返回 false。

boolean ret ;dll.Invoke(ret) 最后，我们调用 EnumWindows 函数。

我们再来看看 ue_callback 这个事件：



首先看它的声明部分，2 个 ulong 类型参数，返回 boolean，与我们定义的回调一致。

下面代码部分，是具体业务实现。这里我们根据传入 的 2 个参数，一个是 HWND，另一个是我们的自定义参数，即一个自定义字符串。代码里通过 HWND 取窗口标题，然后看我们给定的标题是不是包含在里面，如果是由把这个窗口标题添加到 DW 里面。

通过这个方法，可以在只知道窗口标题关键字的情况下，查找到我们需要的窗口。例如查找记事本程序窗口时，它的窗口标题并不固定，可以是“无标题-记事本”、“日志.log-记事本”等等，自定义“记事本”时，可以找到所有记事本窗口。

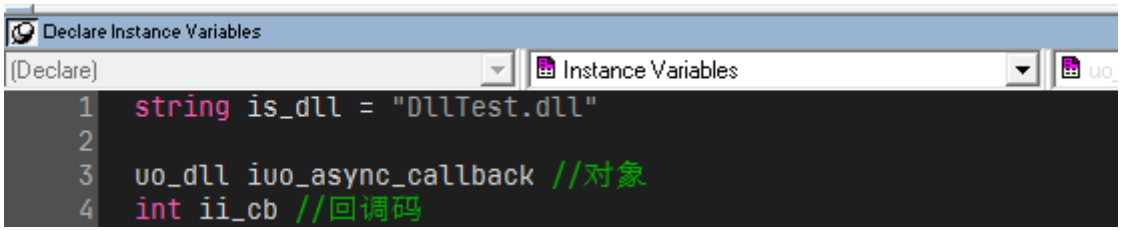
六、跨线程异步回调

其实大多数回调，使用的是跨线程回调。例如海康 SDK，它有许多线程在后台工作，需要时才回调，通知主线程，或者传送数据。

PB 里是没办法直接用线程去响应这种跨线程回调。uo_dll 里采取了线程同步的方式，在回调线程里把数据同步到主线程里，主线程及时保存数据后，让回调线程继续工作。

跨线程异步回调，在实现上与同步回调有少许区别：

1.由于一个函数或事件内声明实例，在函数或事件结束时，会被释放，导致我们创建的对象和回调都被释放，而后台线程在工作，会导致崩溃。解决办法：用进程实例变量保存对象和回调变量。



2.由于内部采取了线程同步机制，线程卡在那等回调事件响应，默认是等 1 秒。通常 1 秒足够我们事件响应了，但我们没必要让线程卡在那白白等 1 秒钟。所以我们事件处理完后，要通知它继续。

```
iuo_async_callback.duang(ii_cb)
//给指定回调敲个钟，让回调线程继续
subroutine Duang(int callback) system library "pbidea.dll" alias for "funcDuang"
```

这个函数就相当敲 一下钟，通知回调线程，我们这处理完了，你继续吧。它的参数是绑定的那个回调值。

以上介绍了 pbidea 之 uo_dll 调用 DLL 函数的方法，解决了 PB 不能调用 __cdecl 约定的函数和回调函数的痛点。

更多多相关知识，可以加入 QQ 群 624409252 进行交流。

同时，强烈谴责佩老棺材瓢子窜访台湾！

大自在,QQ:781770213

2022/8/3